TEXAS INSTRUMENTS

SOFTWARE

TI-74
LEARN PASCAL

Texas Instruments TI–74 Learn Pascal Reference Guide

This book was developed and written by the staff of Texas Instruments Instructional Communications.

Additional corrections and formatting was performed by Jack W. Hsu.

The *TI–74 Learn Pascal Reference Guide* describes the commands, procedures, and functions as well as may of the reserved words that can be used in TI–74 Pascal. The commands, statements, and functions are shown in Extended Backus-Naur Form (EBNF), which uses the following conventions in a syntax description.

- “ ” Any symbol or word within quotes must appear in the program exactly the way it appears within the quotes.
- [ ] Optional elements are enclosed in brackets.
- { } Items enclosed in braces may be used zero or more times.
- ¦ Vertical bars are used to separate options of which only one can be chosen.
- *Italicized* words are names of syntactic elements that you must supply.

The **Description** provides the keyword’s use of function and includes the options that the keyword can use.

The **Example** give the keyword’s use in a short program

Table of Contents

ABS 7
AND 8
ARRAY 9
assignment 10
ATAN 11
BEGIN 12
block 13
BOOLEAN 14
BREAK 15
BYE 16
CASE 17
CHAR 19
CHR 20
CLOSE 21
comment 23
CONCAT 24
CONST 25
CONTINUE 26
COPY 27
COS 28
DEL 29
DELETE 30
DIV 31
DO 32
DOWNTO 33
ELSE 34
END 35
EOF 36
EOLN 37
EXIT 38
EXP 40
FALSE 41
FILE 42
FILLCHAR 43
FOR 45
FORMAT 47
FORWARD 48
FUNCTION 49
GOTO 50
GOTOXY 51
HALT 52
identifier 53
IF 54
IN 56
INPUT 57
INSERT 58

INTEGER 59
INTERACTIVE 60
interpreter options 61
IORESULT 63
KEYBOARD 66
LABEL 67
LENGTH 68
LIST 69
LN 70
LOG 71
MAXINT 72
MEMAVAIL 73
MOD 74
MOVELEFT 75
MOVERIGHT 76
NEW 77
NOT 78
NUMBER 79
ODD 80
OF 81
OLD 82
OR 83
ORD 84
ordinal type 85
OUTPUT 86
PACKED 87
PAGE 88
POS 89
PRED 90
PROCEDURE 91
PROGRAM 92
PWROFTEN 93
READ 94
READLN 96
REAL 99
recursion 100
RENUMBER 101
REPEAT 102
RESET 104
REWRITE 106
ROUND 107
RUN 108
SAVE 109
SCAN 110
SIN 112
SIZEOF 113
SQR 114
SQRT 115
statement 116
STR 117

STRING 118
subrange 119
SUCC 120
TEXT 121
THEN 122
TO 123
TRUE 124
TRUNC 125
TYPE 126
UNBREAK 127
VAR 128
VERIFY 129
WHILE 130
WRITE 131
WRITELN 133

Appendix A—Commands 137
Appendix B—Functions 138
Appendix C—Statements 139
Appendix D—Procedures/Functions 140
Appendix E—Declarations and Data Types 141
Appendix F—Reserved Words 142
Appendix G—ASCII Codes and Keycodes List 143
Appendix H—Accuracy Information 155
  Calculation Accuracy 156
  Internal Numeric Representation 157
Appendix I—Error Messages 158
  Messages Listed Alphabetically 160
  Messages Listed in Numerical Order 164
  I/O Error Codes 167
Service Information 170
One-Year Limited Warranty 172

ABS

Syntax “ABS” “(“ *integer-expression* ¦ *real-expression* “)”

Description The ABS function returns the absolute value of *integer-expression* or *real-expression*. The value returned is of the same type as the expression. If *integer expression* or *real-expression* is positive or zero, ABS returns the value of the expression. If *integer-expression* or *real-expression* is negative, ABS returns the negative of the expression. The result of ABS is always positive or zero.

Example The following program multiplies 4 by the absolute value of –4, displays the result, and then displays the absolute value of –7.12.

```
100 PROGRAM EXABS;
110 VAR rval:REAL;
120 BEGIN
130   WRITELN(4*ABS(-4);
140   rval:=ABS(-7.12);
150   WRITELN('The value is',rval);
160 END. (* exabs *)
```

Output

```
16
The value is 7.12
```

AND

Syntax *Boolean-expression1* “AND” *Boolean-expression2*

Description The Boolean operator AND performs the logical conjunction of two Boolean expressions. The order of precedence for logical operators from highest to lowest is NOT, AND, and OR. The operation AND is performed on the same precedence level as the arithmetic operators *, /, DIV, and MOD, before the operators +, –, and OR, and before the relational operators =, <>, <, <= , >, >= , and IN.

The following table gives the results of performing an AND on the possible combinations of Boolean expressions.

Examples

Booexpl/Booexp2	**TRUE**	**FALSE**
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

ARRAY

Syntax [“PACKED”] “ARRAY” “[“ *ordinal-constant* “..” *ordinal-constant* { “,” *ordinal-constant* “..” *ordinal-constant* } “]” “OF” *type*

Description An ARRAY is a structured type that has a specified number of elements, all of which have the same type. The type of each element is called the base type of the array and can be any type except file.

An array can have one, two, or three dimensions. Each element in an array is accessed by the array identifier and an index enclosed in brackets. The index type of an array is the data type of the index and must be INTEGER or CHAR.

An array identifier can be used as shown below.

array-identifier “[“ *expression* { “,” *expression* } “]”
¦
array-identifier “[“ *expression* { “]” “[“ *expression* } “]”

The following are examples of array references.

```
intarray[5]
realarray[2,3]
strarray['D']['G']
charray[k+5,m-3,n+2]
```

A reference to an array must contain the same number of indices as was declared in the array’s declaration, except that a STRING array can be referenced with one more index and a PACKED ARRAY OF CHAR with one less index.

assignment

Syntax

variable ":=" *expression*
function-identifier ":=" *expression*

Description

The assignment statement is used to assign a value to a variable or to assign a value to a function when it is called. The type of the identifier to the left of := must be the same type as the expression on the right (except that INTEGER values can be assigned to REAL variables).

Examples

```
n:=78.4;
count:=count+1;
name:='hello';
grade:='A';
flag1:=TRUE;
realval:=78;
```

ATAN

Syntax “ATAN” “(“ *integer-expression* ¦ *real-expression* “)”

Description The ATAN function returns the measure in radians of the angle whose tangent is *integer-expression* or *real-expression*. The value returned by ATAN is a real value between –π/2 and π/2.

Example The following program reads a number from the keyboard and displays the arctangent of the number.

```
100 PROGRAM exatan;
110   CONST pi=3.14159265359;
120   VAR tang,angle:REAL;
130 BEGIN
140  WRITE('Enter tangent: ') {$w-};
150   READLN(tang {$w+};
160   angle:=ATAN(tang);
170   WRITELN('Atan of ',tang:5:2,'=',
       angle:5:2,' rad ’,angle*180/pi:7:2,
       CHR(223));
180 END. (" exatan ")
```

Input:

```
7.12
```

Output:

```
Atan of  7.12=1.43 rad 82.01°
```

BEGIN

Syntax “BEGIN” [*statement* { “;” *statement* }] “END”

Description The reserved word BEGIN is used with the reserved word END to enclose the body of a block and also to delimit a group of statements into one compound statement.

The first BEGIN in the body of a block signals the Pascal interpreter of the end of the declaration section of a block and the beginning of the executable statements. Any other BEGIN in this block is used with the reserved word END to enclose several statements into a single compound statement.

Example BEGIN is used in all program examples throughout this manual.

block

Syntax

program heading
[*label declaration*]
[*constant declaration*]
[*type declaration*]
[*variable declaration*]
[*procedure* ¦ *function declaration*]
"BEGIN" [*statement* { ";" *statement* }] "END."
¦
procedure ¦ *function heading*
[*label declaration*]
[*constant declaration*]
[*type declaration*]
[*variable declaration*]
"BEGIN" [*statement* { ";" *statement* }] "END" ";"

Description

A block consists of a heading, a sequence of declarations, and a sequence of statements enclosed by the reserved words BEGIN and END. A program block can contain other blocks such as procedure and function blocks.

An identifier defined in the declaration section of a program is called a global constant or variable and can be accessed from any block in the program. A identifier defined in a procedure or function block is called a local variable and can be accessed only within the procedure or function where it is declared.

BOOLEAN

Syntax "BOOLEAN"

Description The type BOOLEAN is a predefined ordinal type that is used to represent truth values. A BOOLEAN type is either TRUE or FALSE. FALSE is defined to be 0 and TRUE is defined to be 1.

The logical operators NOT, AND, and OR can be used with BOOLEAN types. The relational operators =, <>, <, <=, > >=, and IN can be used with a valid type to return a BOOLEAN value.

The following functions can be used to return a BOOLEAN value.

EOF
EOLN
ODD
PRED
SUCC

BOOLEAN types cannot appear in input/output statements.

BREAK

Syntax “BREAK” *line-number* { “,” *line-number* }

Description The BREAK command is used to halt program execution at specific points, called breakpoints. *Line-number* is the number of a program line. When the BREAK command is entered, a breakpoint is set at the beginning of the first executed statement following each specified *line-number*. When a breakpoint occurs, the message `Break` is displayed.

The BREAK command is useful in debugging a program. When a breakpoint halts execution of a program, the massage `Break` is erased when the CLR or ENTER key is pressed. The values of variables can then be displayed and calculations can be performed to determine why a program is not executing properly. The CONTINUE command can be used to resume program execution (except within a user-defined function) if the program has not been edited.

Note that the BREAK key also causes the program to stop executing as if a breakpoint had been encountered.

A breakpoint remains in a program until the UNBREAK command is used to remove it or until the line is edited or deleted.

Example

```
BREAK 150,210,450
```

causes breakpoints to be set at lines 150, 210, and 450.

BYE

Syntax “BYE”

Description The command BYE is used to leave the Pascal system and return control to the BASIC command level.

Note: When the BYE command is executed, the BASIC system is initialized and any Pascal program in memory is erased.

Example

```
BYE
```

returns to the BASIC command level. The BASIC system is initialized and ready to accept commands and statements.

CASE

Syntax

"CASE" *ordinal-expression* "OF" *constant* { "," *constant* } ":" *statement* { ";" *constant* { "," *constant*} ":" *statement* } "END"

Description

The CASE statement is used to select one of several alternatives. The alternatives are the constants after the reserved word OF. When a CASE statement is executed, *ordinal-expression* is evaluated. *Constant* and *ordinal-expression* must be of the same type. If the value of *ordinal-expression* is the same as a *constant* listed, the *statement* following that *constant* is executed.

In the series of alternatives, more than one *constant* can precede *statement*. Each *constant* in a CASE statement should appear only once; otherwise, only the first use of the *constant* is implemented.

If the value of *ordinal-expression* is not equal to any of the constants, program execution continues with the statement following the CASE statement.

Note that these statement labels are not the same as those defined with a LABEL declaration. The labels of a CASE statement cannot appear in a GOTO statement.

Example The following program reads characters entered from the keyboard. If this character is a digit from 1 through 9, the appropriate suffix is displayed after the digit. The program ends when a character is entered that is not a digit from 1 through 9.

```
100 PROGRAM excase;
110 VAR ch:CHAR;
120   suffix:STRING;
130 BEGIN
140   READLN(ch);
150     WHILE ch IN['1'..'9'] DO
160       BEGIN
170         CASE ch OF
180           '1':suffix:='st';
190           '2':suffix:='nd';
200           '3':suffix:='rd';
210           '4','5','6','7','8','9':
               suffix:='th';
220         END;
230         WRITELN(ch,suffix);
240         READLN(ch)
250       END;(* while ch is valid *)
260 END. (* excase *)
```

CHAR

Syntax “CHAR”

Description The type CHAR is a predefined ordinal type that represents individual characters. The characters are ordered according to their ASCII codes as shown in appendix G. The relational operators can be used to compare CHAR data. The functions ORD, PRED, and SUCC can be sued with CHAR data to determine the ordinal number (ASCII code) of a character, its predecessor, and its successor, respectively.

CHR

Syntax "CHR" "(" *integer-expression* ")"

Description The CHR function returns the character corresponding to the ASCII character code specified by *integer-expression*. A list of the ASCII character codes for the TI-74 character set is given in appendix G.

If integer-expression evaluates to a number greater than 255 it is repeatedly reduced by 256 until the value is between 0 and 255, inclusive. If integer-expression evaluates to a number less than 0, it is repeatedly increased by 256 until the value is between 0 and 255 inclusive.

Example The following program displays all of the characters listed in appendix G as displayed by the CHR function. The ENTER key can be pressed to display each successive character or can be held down for a more rapid display.

```
100 PROGRAM exchr;
110 VAR count:INTEGER;
120 BEGIN
130   FOR count:=0 TO 255 DO
140   WRITELN(count,' ',CHR(count));
150 END. (* exchr *)
```

CLOSE

Syntax “CLOSE” “(” *file-identifier* [“,” “LOCK” ¦ “PURGE”] “)”

Description The CLOSE procedure closes the file specified by *file-identifier*. A closed file cannot be accessed in an program until a RESET or REWRITE statement is executed. The effect of using CLOSE to close a file is dependent upon how the file was opened, as shown below.

	File opened with:	
	RESET	**REWRITE**
CLOSE(*file-identifier*)	closes the file	deletes the file
CLOSE(*file-identifier*,LOCK)	closes the file	closes the file
CLOSE(*file-identifier*,PURGE)	deletes the file	deletes the file

When the Pascal program finishes executing, the interpreter automatically closes any open files.

Example

The following program opens files for output on a mass-storage device specified as device 1. The number of strings to be entered from the keyboard is then input. After a string is read, it is written to device 1. After all the strings are entered, the file `data10` is closed and reset to the first element in the file. The strings are then read from `data10` and printed on device 10.

```
PROGRAM exclose;
VAR filein:TEXT ;
  filepr:TEXT ;
  count,index:INTEGER;
  st1:STRING;
BEGIN
  REWRITE(filein,'l.data10');
  REWRITE(filepr,'10');
  WRITE('Enter # of entries: ') {$w-};
  READLN(count) {$w+};
  FOR index:=1 TO count
    BEGIN
      READLN(st1);
      WRITELN(filein, stl);
    END;(* for loop *)
  CLOSE(filein,LOCK);
  RESET(filein,'l.data10');
  FOR index:=1 TO count
    BEGIN
      READLN(filein,st1) ;
      WRITELN(filepr,st1);
    END;(* for loop *)
END. (* exclose *)
```

comment

Syntax “{“ sequence of characters that does not contain closing delimiter of comment “}”
¦
“(*” sequence ofcharacters that does not contain closing delimiter of comment “*)”

Description A comment is used within a program to document and explain sections of a program. A comment is any text written between { and } or (* and *). The symbols used to enclose a comment cannot be mixed, but one comment may contain a nested comment enclosed in the other set of symbols as shown below.

```
{WRITELN(counter);(* loop counter *)}
```

A comment can be placed at any location in a program where a space can occur. In TI-74 Pascal, a comment may not be extended over multiple lines.

Comments can be used to specify options that are to be implemented by the interpreter as a program is executing. An option is implemented when a $ and the letter a, i, or w immediately follow the opening delimiter of a comment. A plus sign (+) after the letter causes the interpreter to turn on the option. A minus sign (-) after the letter causes the interpreter to turn off the option. For more information, refer to **interpreter options**.

CONCAT

Syntax "CONCAT" "(" *string-expression* {"," *string-expression*} ")"

Description The CONCAT function returns the string formed when all strings listed are linked together or concatenated. If the number of characters in the strings to be linked is greater than 255, CONCAT displays the message `Truncation warning` and truncates the string to the first 255 characters. If a string variable is assigned a concatenated string that is longer than the declared length for the string variable, the message `Truncation warning` is displayed.

Example The following program concatenates the three strings `st1`, `st2`, and `st3` and then displays the concatenated string.

```
100 PROGRAM exconcat;
110 VAR stn,st1,st2,st3:STRING;
120 BEGIN
130   stl:='CONCAT ';
140   st2:='returns a con';
150   st3:='catenated string';
160   stn:=CONCAT(st1,st2,st3);
170   WRITELN(stn);
180 END. (* exconcat *)
```

Output:

```
CONCAT returns a concatenated string
```

CONST

Syntax

"CONST" *identifier* "=" *constant* ";" { *identifier* "=" *constant* ";")

where constant is

["+" ¦ "–"] *constant-identifier* ¦ *unsigned-number* ¦ *predefined-constant* ¦ *character* { "," *character*}

Description

The CONST declaration is used to associate an identifier with a data value. This data value cannot change during program execution. The type of value assigned to an identifier in a CONST declaration determines the type of the identifier.

A string constant has a fixed length. An identifier defined to be one character is a CHAR type and must be used accordingly throughout the program. A numeric constant may be declared as the opposite of a previously declared numeric constant. Other than this use, no expressions are allowed in a CONST declaration.

Examples

```
CONST pi=3.14159265359;(* REAL *)
      col=12;(* INTEGER *)
      row=20;(* INTEGER *)
      msg='Refer to page';(* STRING *)
      letter='z';(* CHAR *)
      rval=5.5;(* REAL *)
      negrval=-rval;(* REAL *)
      posrval=rval;(* REAL *)
      bool=TRUE;(* BOOLEAN *)
```

CONTINUE

Syntax "CONTINUE" ¦ "CON"

Description The CONTINUE (or CON) command is used to resume program execution after a breakpoint has occurred. Program execution resumes at the statement following the breakpoint.

The following actions do not allow a CONTINUE command to resume program execution.

- Editing the program.
- Entering a NEW, OLD, RENUMBER, RUN, or SAVE command.
- Listing the program to a peripheral device.
- Turning the system off or pressing the RESET key.
- Breaking within a function.

Note: If the BBEAK key is pressed while data is being entered for an input statement, CONTINUE re-executes the input statement.

Example

```
CON
```

resumes execution of the program in memory.

COPY

Syntax

"COPY" "(" *string-expression* "," *integer-expression*1 ","
integer-expression2 ")"

Description

The COPY function returns a string that is a substring of another string. *Integer-expression1* is the index position in *string-expression* where the substring begins. *Integer-expression2* specifies the number of characters to copy into the substring.

The characters in a string are indexed from 1 through the dynamic length of the string. The dynamic length of a string cannot be greater than the declared length (or default length if one is not specified) of the string.

Both of the integer expressions must be positive integers. If *integer-expression2* specifies more characters than there are in the string starting from *integer-expression1*, no characters are copied into the substring.

Example

The following program copies part of `st1` into `st2` and then displays the copied string.

```
100 PROGRAM excopy;
110 VAR stl,st2:STRING;
120 BEGIN
130   stl:='The default length of a
       string is 80';
140   st2:=COPY(stl,5,26);
150   WRITELN(st2);
160 END. (* excopy *)
```

Output:

```
default length of a string
```

COS

Syntax "COS" "(" *integer-expression* ¦ *real-expression* ")"

Description The COS function returns the trigonometric cosine of the angle whose measurement in radians is *integer-expression* or *real-expression*. *Integer-expression* or *real-expression* must be a number whose absolute value is less than $\pi/2*10^{10}$. The value returned by COS is a real value.

Example The following program computes the number of squares of shingles it requires to cover a roof. The pitch of the roof the length and width of the house are entered and the area the roof is computed. The area is divided by 100 and then rounded before it is displayed.

```
100 PROGRAM excos;
110 (* computes # squares to cover roof *)
120 (* square = 100 square feet *)
130 CONST pi=3.14159265359;
140 VAR angle,lenth,width:REAL;
150     squares:INTEGER;
160 BEGIN
170   WRITE('Enter pitch of roof (deg): ' ) {$w-};
180   READLN(angle);
190   WRITE('Enter length and width (ft): ');
200   READLN(lenth,width) {$w+};
210   angle:=angle*pi/180;
220   IF COS(angle)<=0.0
230     THEN WRITELN(' invalid pitch')
240     ELSE
250       BEGIN
260         squares:=ROUND(width/COS(angle)*lenth/100);
270         WRITELN('Area of roof: ',squares,'squares');
280       END;
290 END. (* excos *)
```

Input:

```
75
60 40
```

Output:

```
Area of roof: 93 squares
```

DEL

Syntax “DEL” *line-number-list*
¦
“DEL” “,” *device* “.” *filename* “.”

Description The DEL command is used to remove lines from a program in memory or to remove a file from external storage. *Line-number-list* may be anyone or any combination of the following (separated by commas).

Line-number-list	**Action**
a single line number	deletes that line
line number-	deletes that line number and all succeeding lines
-line number	deletes that line number and all preceding lines
line number -line number	deletes that inclusive range of lines

If a line number is specified that does not exist, the message `Line not found` is displayed. However, any remaining lines specified are deleted when the ENTER or CLR key is pressed. If the initial line of a range does not exist, the next higher-numbered line is used as the initial line. If the final line does not exist, the next lower-numbered line is used as the final line.

Device.filename is used to delete a file from an external storage device. *Device* is the number associated with the peripheral device and can be from 1 through 255. *Filename* identifies a particular file that must be closed. Refer to the peripheral manuals for the device number of a device and for specific information about the form of *filename*.

Example

```
DEL 150,170,370-
```

deletes lines 150, 170, and lines 370 through the end of the program currently in memory.

```
DEL '1.file10'
```

deletes the file named file10 from device 1.

DELETE

Syntax “DELETE” “(“ *string-variable* “,” *integer-expression1* “,” *integer-expression2* “)”

Description The DELETE procedure removes a specified number of characters from the string specified by *string-variable*. *Integer-expression1* is a positive integer that specifies the index position in *string-variable* where the deletion begins. *Integer-exprression2* is a positive integer that specifies the number of characters to delete. If *integer-expression2* specifies more characters than DELETE can remove, no characters are deleted.

The following program deletes part of `st1` and then displays the modified string.

Example

```
100 PROGRAM exdelete;
110 VAR st1:STRING;
120 BEGIN
130   st1:='DELETE changes the dynamic length of a string';
140   DELETE(st1,16,22);
150   WRITELN(st1);
160 END. (* exdelete *)
```

Output:

```
DELETE changes a string
```

DIV

Syntax *integer-expression* “DIV” *integer-expression*

Description The arithmetic operator DIV is used between two INTEGER type operands to obtain the integer portion of their quotient. Any fractional portion is discarded. If the second operand has a value of zero, an error occurs.

The DIV operator is on the same precedence level as the operators AND, “/”, and MOD.

Examples `5 DIV 2` is `2`

`14 DIV 2` is `7`

Syntax Refer to the FOR and WHILE statements

Description The reserved word DO is part of the FOR and WHILE statements. DO is written before the statement that is to be executed in the FOR–or the WHILE–loop. If multiple statements are to be executed in the loop, they are in enclosed in the reserved words BEGIN and END. If a semicolon is placed immediately after the reserved word DO, no statement is executed, but the FOR or WHILE statement is repeated until the condition specified in the statement is satisfied.

DOWNTO

Syntax Refer to the FOR statement

Description The reserved word DOWNTO is used in the FOR statement to cause the control variable to take consecutive preceding values in the variable's defined set of values. For INTEGER types, the control variable is decreased by one each time through the FOR loop. For a CHAR type, the control variable becomes the preceding character each time through the loop.

ELSE

Syntax Refer to the IF statement

Description The reserved word ELSE is an optional part of the IF statement. When an IF statement contains an ELSE part, the statement following the word ELSE is executed when the Boolean expression of the IF statement is FALSE. If more than one statement is to be executed, the statements must be enclosed in the reserved words BEGIN and END.

With nested IF statements, an ELSE is paired with the IF that has not been matched.

Note that if a semicolon immediately precedes the reserved word ELSE, an error occurs.

END

Syntax

"BEGIN" [*statement* {";"*statement*}] "END"
¦
"CASE" *ordinal-expression* "OF"
    constant {"," *constant*} ":" *statement* {";"
    constant {"," *constant*} ":" *statement*}
"END"

Description

The reserved word END terminates a block, a compound statement, or a CASE statement. END is used with the word BEGIN to delimit a block or a compound statement. When END terminates a main program body, END must be followed by a period.

Example

The following program reads five integers. If the integer is from 1 through 5, its English equivalent is displayed.

```
100 PROGRAM exend ;
110 VAR count,intval : INTEGER;
120 BEGIN (* begins executable section *)
130   FOR count:=l TO 5 DO
140     BEGIN (* forms a loop *)
150       WRITE(' Enter value (1-5): ')
           {$w-};
160       READLN(intval) {$w+};
170       IF intval IN[1..5]
180         THEN CASE intval OF
190           l:WRITELN('one');
200           2:WRITELN('two');
210           3:WRITELN('three');
220           4:WRITELN('four');
230           5:WRITELN('five');
240           END;(* case statement *)
250     END;(* End for loop of 5 values *)
260 END. (* exend *)
```

EOF

Syntax “EOF” [“(” *file-identifier* “)”]

Description The EOF function determines if the next data read from a text file will be an *end-of-file* marker. EOF is true if the *end-of-file* marker is the next character to be read on a text file: otherwise, EOF is false. There is no EOF marker for an INTERACTIVE file.

File-identifier is the identifier of an open file. If *file-identifier* is omitted, the file INPUT is assumed.

Example The following program reads a file stored on a mass-storage device and then prints the file. When an end-of-line mark on the file has been read, an end-of-line marker (WRITELN) is sent to the printer. When the end-of-file marker is reached the file is closed.

```
PROGRAM exeof ;
VAR file1,filepr:TEXT;
    ch:CHAR ;
BEGIN
  RESET(file1,'1.file9');
  REWRITE(filepr,'50');
  WHILE NOT EOF(file1) DO
    BEGIN
      WHILE NOT EOLN(fi1e1) DO
        BEGIN
          READ(file1,ch);
          WRITE(filepr,ch);
        END;(* EOLN test *)
      READLN(file1);
      WRITELN(filepr);
    END;(* EOF test *)
  CLOSE(file1,LOCK);
  CLOSE(fiIepr,LOCK);
END. (* exeof *)
```

EOLN

Syntax "EOLN" ["(" *file-identifier* ")"]

Description The EOLN function determines if the next character to be read in a text file is the end-of-line marker or if the end-of-line marker has been read on an INTERACTIVE file. *File-identifier* is the identifier of an open file. If *file-identifier* is omitted, the file INPUT is assumed.

If *file-identifier* specifies a TEXT file, EOLN returns a TRUE result when the next character to be read is the end-of-line marker or when EOF is TRUE. If *file-identifier* specifies an INTERACTIVE file, EOLN returns a TRUE result when the end-of-line marker was the last character read.

If an input statement attempts to read data when the end-of-line marker is the next character to be read from a text file, the value of the variable in the input statement is dependent upon the type of variable being read. A CHAR type variable is assigned a blank; a STRING type variable is assigned a null string. For INTEGER and REAL types, the computer reads past the EOLN marker until it encounters the first nonblank character.

Example The following program accepts characters from the keyboard and prints them until an asterisk is read.

```
100 PROGRAM exeoln;
110 VAR file1:INTERACTIVE;
120     filepr:TEXT;
130     ch:CHAR ;
140 BEGIN
150   REWRITE(filepr,'10');
160   READ(file1,ch);
170   WHILE ch<>'*' DO
180     BEGIN
190       IF EOLN(file1)
200         THEN WRITELN(filepr)
210         ELSE WRITE(filepr,ch);
220       READ(file1,ch);
230   END;(* EOF test *)
240 END. (* exeoln *)
```

EXIT

Syntax "EXIT" "(" "PROGRAM" ¦ *program-identifier* ¦ *procedure-identifier* ¦ *function-identifier* ")"

Description The EXIT procedure can be used to terminate the execution of a program or a procedure or function. When EXIT is used terminate a program, either the *program-identifier* or the reserved word PROGRAM can be specified. All open files closed.

When EXIT is used to terminate a procedure or a function, *procedure-identifier* or *function-identifier* must be specified. The *procedure-identifier* or *function-identifier* specified does not have to be the routine that contains the EXIT procedure. When the EXIT statement is executed, the interpreter terminates the routine containing the EXIT statement and each calling routine back to and including the one specified in the EXIT statement. Each open local file is closed.

If the procedure or function identifier passed to EXIT is a recursive routine, only the last call to the routine is exited.

If an EXIT procedure terminates a function before an assignment has been made to *function-identifier*, the function returns an undefined value.

A program, procedure, or function automatically terminated when its END statement is encountered; therefore, EXIT should be used only in exceptional cases.

Note that EXIT cannot be used as an imperative.

Example

The following program requires that the code *123/4* be entered before the program continues execution. If the correct code is not entered, the program terminates with an EXIT. Otherwise, the file `file9` is read from a mass-storage device and printed.

```
100 PROGRAM exexit;
110 VAR file1,filepr:TEXT;
120     codestr,strdata:STRING;
130 BEGIN
140   WRITE('Enter code to use program: ')
       {$w-};
150   READLN(KEYBOARD,codestr) {$w+};
160   IF codestr<>'*123/4*'
170     THEN EXIT(PROGRAM);
180   RESET(file1,'1.file9');
190   REWRITE(filepr,'20');
200   WHILE NOT EOF(file1) DO
210     BEGIN
220       READLN(file1,strdata);
230       WRITELN(filepr,strdata);
240     END;(* EOF test *)
250 END. (* exexit *)
```

EXP

Syntax “EXP” “(” *integer-expression* ¦ *real-expression* “)”

Description The EXP function returns the result of e^x, where x is the value of *integer-expression* or *real-expression*. The value of e is 2.71828182846. The value returned by EXP is a real value .

Example The following program accepts two real numbers and displays the first number raised to the power specified by the second number. The program then prompts `Continue? (y or n)` and reads the response from the KEYBOARD file, which do not echo to the display. If `y` is pressed, the program repeats. If `n` is pressed, the program terminates.

```
PROGRAM exexp;
VAR baseval,expval:REAL;
    ch:CHAR;
BEGIN
  REPEAT
    WRITE('Enter base # and its power: ')
     {$w-};
    READLN(baseval,expval) {$w+};
    WRITELN(baseval,CHR(94),expval,'=',
      EXP(expval*LN(baseval));
    WRITE('Continue? (y or n) ') {$w-};
    READLN(KEYBOARD,ch) {$w+};
  UNTIL NOT(ch='Y') OR (ch='y');
END. (* exexp *)
```

FALSE

Syntax “FALSE”

Description The predefined BOOLEAN constant FALSE is equal to the BOOLEAN value FALSE. FALSE is defined to be less than TRUE. The ordinal value of FALSE is 0.

FILE

Syntax *file-identifier* ":" ."TEXT"
|
file-identifier ":" *user-defined-file-type*

Description A FILE is a structure consisting of a series of data items that are of the same type. Files are organized sequentially, that is they must be read in the same order they were written. In TI-74 Pascal, a file is written in ASCII characters (called a TEXT file).

A file consists of records, which are fields of data. The fields of data correspond to the values of items listed in an output statement. At the end of each record is an end-of-line marker. The end-of-file marker is written after the last end-of line marker on a TEXT file.

When a file is opened, a buffer is created in memory large enough to hold any record of the file. The maximum length of a record is dependent on the device containing the file. A record can be shorter than the maximum allowed. The length of a record is the number of characters written by an output statement(s). The buffer that is created when a file is opened is the only means of data transfer from a file to a program in memory.

Before data can be transferred to or from a file, the file must be opened with *file-identifier* in a RESET or REWRITE statement. RESET opens an existing file for input only; REWRITE opens a file for output only. If the file does not already exist, REWRITE creates a file containing only the end-of-file marker. If the file already exists, REWRITE deletes the existing file and creates a new file containing the end-of-file marker.

The procedure CLOSE is used to close a file.

FILLCHAR

System "FILLCHAR" "(" *variable* "," *integer-expression* "," *character-expression* ")"

Description The FILLCHAR procedure fills a specified number of bytes with a specified character. Variable can be any type except a file type. *Integer-expression* specifies the number of bytes to fill starting at the location specified by variable. *Character-expression* specifies the character to be stored in the bytes.

Note: If variable is a string variable, *variable* should be indexed. Otherwise, the FILLCHAR procedure begins at the length byte of the string (rather than one of its characters).

Warning: Use care when filling bytes with FILLCHAR. If system data is written over, the computer may have to be reset to continue operation.

Example The following program uses FILLCHAR to assign the plus sign (+) to the 80 elements of the array `ch`. FILLCHAR assigns the # sign to the array `str1`. The number of # signs assigned is determined by the function SIZEOF, which determines the number of bytes that the array str1 uses in memory. FILLCHAR assigns an asterisk (*) to the string `str2` starting at the first character position in the string. The number of asterisks assigned is determined by subtracting one from the number of bytes the string `str2` uses in memory. A string always uses the same number of bytes as there are characters currently assigned to the string plus one byte that contains the length of the string.

The arrays `ch` and `str1` and the string `str2` are then displayed.

```
PROGRAM exfillch;
VAR ch:PACKED ARRAY[l..80] OF CHAR;
    str1:ARRAY[1..80] OF CHAR ;
    str2:STRING;
    index: INTEGER;
BEGIN
  str2:='FILLCHAR will write over this string';
  FILLCHAR(ch,80,'+');
  FILLCHAR(str1,SIZEOF(str1),'#');
  FILLCHAR(str2[1],LENGTH(str2),'*');
  WRITELN(ch);{$w-}
  FOR index:=l TO SIZEOF(str1) DO
    WRITE(str1[index]);{$w+}
  WRITELN;
  WRITELN(str2);
END. (* exfillch *)
```

Output:

```
+++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++++++++++++++
###########################################
#####################################
************************************
```

FOR

Syntax

"FOR" *index-variable* ":=" *start-expression* "TO" *end-expression* "DO" *statement*
¦
"FOR" *index-variable* ": =" *start-expression* "DOWNTO" end-expression "DO" *statement*

Description

The FOR statement repeats *statement* a specified number of times. The types of *index-variable*, *start-expression*, and *end-expression* must be the same (usually INTEGER). When a FOR statement is first executed, *start-expression* and *end-expression* are evaluated. *Index-variable* is then assigned the value of *start-expression* and statement following the reserved word DO is executed.

If TO is used, *index-variable* is assigned the next value in its type, e.g., for integers, *index-variable* is incremented by 1; for characters, *index-variable* is assigned the next character. This sequence is repeated until the value of *index-variable* is greater than *end-expression*.

If DOWNTO is used, *index-variable* is assigned the previous value in its type. This sequence stops when the value of *index-variable* is less than *end-expression*.

If *start-expression* equals *end-expression*, statement is executed one time. If *start-expression* is greater than *end-expression* (or less than when DOWNTO is used), *statement* is not executed.

Statement can be a single statement or a group of statements enclosed in the reserved words BEGIN and END to form a single compound statement. Note that if a semicolon is placed immediately after the reserved word DO, *statement* is not executed in the loop.

Index-variable cannot be modified within the FOR-loop. *Index-variable* must be a local variable and cannot be a REAL or STRING type or a VAR parameter. After a FOR statement finishes executing, *index-variable* is undefined.

Example The following program reads 80 characters from the keyboard and displays the characters in reverse order.

```
100 PROGRAM exfor;
110 VAR count:INTEGER;
120     ch:ARRAY[1..80] OF CHAR;
130 BEGIN
140   FOR count:=1 TO 80 DO
150     READ(ch[count]); {$w-}
160   READLN;
170   FOR count:=80 DOWNTO 1 DO
180     WRITE(ch[count]); {$w+}
190   WRITELN;
200 END. (* exfor *)
```

FORMAT

Syntax "FORMAT" device

Description The FORMAT command initializes the current medium on an external storage device. Formatting a storage medium destroys all previously stored data.

Device is the number associated with each peripheral device and can be from 1through 255. Refer to the peripheral manuals to obtain the device code for each peripheral device.

Example

```
FORMAT 110
```

Initializes the medium currently in the mass-storage device specified as device 110. All data previously stored on the medium is destroyed.

Note: Tapes used with a cassette recorder do not require initialization with the FORMAT command.

FORWARD

Syntax

"PROCEDURE" *identifier* ["(" *parameter-list* ")"] ";"
"FORWARD" ";"
¦
"FUNCTION" *idcntifier* ["(" *parameter-list* ")"] : *type* ";"
"FORWARD" ";"

Description

The FORWARD declaration is used when it is necessary to declare a procedure or function before it is defined. When procedure or function is declared in a FORWARD declaration, its *parameter-list* (and for functions, its *type* declaration) appears only in the FORWARD declaration, not in the PROCEDURE or FUNCTION definition itself.

Example

The following program reads an integer from 1 through 7 and displays the value of that element in the Fibonacci sequence. A Fibonacci sequence is defined to be a sequence of numbers such that each consecutive number is equal to the sum of the two preceding numbers. Because each function in the program calls the other function, a FORWARD declaration must be used to declare one of the functions before it can defined.

```
100 PROGRAM exforwrd;
110 VAR fibnum: INTEGER;
120 FUNCTION addfib(index:INTEGER):INTEGER;FORWARD;
130 FUNCTION fib(count:INTEGER):INTEGER;
140   BEGIN
150     IF count>l
160       THEN fib:=addfib(count)
170       ELSE IF count=l
180         THEN fib:=1
190         ELSE fib:=0;
200   END;(* fib *)
210 FUNCTION addfib:
220   VAR fibl,fib2: INTEGER;
230   BEGIN
240     fibl:=fib(index-l);
250     fib2:=fib(index-2);
260     addfib:=fibl+fib2;
270   END;(* addfib *)
280 BEGIN
290   REPEAT
300     WRITE( {$w-} 'Enter integer (1-7): ');
310     READLN(fibnum);{$w+};
320   UNTIL fibnum IN[1..7];
330   WRITELN('Fibonacci# ',fibnum,' is ',fib(fibnum));
340 END. (* exforwrd *)
```

FUNCTION

Syntax “FUNCTION” *identifier* [“(” *parameter-list* “)”] “:” *type* “;” block
¦
“FORWARD” “;”

where *parameter-list* is

[VAR] *identifier* { “,” *identifier*} “:” *type* { “;” [VAR] *identifier* { “,” *identifier* } “:” type }

Description A function is a block within a program that is declared in a FUNCTION declaration with an *identifier*. *Identifier* is used in expressions to obtain a value. The type of the value returned by a function must be declared in *type*. Any *identifiers* defined within a function are local to that function. The function can use identifiers defined in the program block.

The statements in a function are performed when an expression containing the function identifier is executed. Some unallocated part of memory is used for the function's local variables when its statements are executed. A function must execute at least one statement that assigns the function identifier a value; otherwise, the function is undefined. After the function terminates, the memory used for the local variables is released.

A FUNCTION declaration may also contain parameters to transfer data between a function and the block in which it is defined. The parameters can be value parameters or VAR parameters.

A parameter that is a value parameter is a local variable within the function. When *identifier* is called, the values of the parameters in the function call are evaluated and assigned to the corresponding value parameters in the function. Any change made to a value parameter is not made to the corresponding parameter in the function call.

A parameter that is a VAR parameter uses the storage location of the corresponding parameter in the function call. Any change made to a VAR parameter is also made to its corresponding parameter in the function call because they use the same memory locations.

A function can be recursive. Refer to FORWARD for an example.

GOTO

Syntax “GOTO” *label*

Description The GOTO statement transfers program execution to the Iine specified by *label*. *Label* must be an unsigned integer from 0 through 9999 that has been declared in a LABEL declaration. Any declared *label* must appear before one (and only one) statement in the program. GOTO can transfer control to any statement within its own block, except a statement in a loop. GOTO cannot be used in an imperative.

Generally, GOTO statements should be avoided in Pascal programs because they obscure the structure of the program.

Example The following program reads an integer from the keyboard. If the integer is greater than 25, a branch to label 300 is executed. If the integer is less than 21, a premium of 100 is displayed. Otherwise, a premium of 80 is displayed.

```
PROGRAM exgoto;
LABEL 300;
VAR age:INTEGER;
    special,rate:REAL;
BEGIN
  special:=0.0;
  WRITE('Enter age: ') {$w-};
  READLN(age) {$w+};
  IF age>25
    THEN GOTO 300;
  IF age<21
    THEN special:=100.0
    ELSE special:=80.0;
  WRITELN('Pay premium of $',special:6:2,' plus ');
  300:WRITELN('Pay amount shown on bill');
END. (* exgoto *)
```

GOTOXY

Syntax “GOTOXY” “(” *integer-expression*1 “,” *integer-expression2* “)”

Description The GOTOXY procedure moves the cursor to the location specified by its parameters. *Integer-expression1* specifies the column position, the first column being defined as column 0. *Integer-expression2* specifies the row position, the first row being defined as row 0. In TI-74 Pascal, the row specification must be 0.

If either *integer-expression1* or *integer-expression2* specifies an invalid position, the warning `Implementation restriction` is displayed and the invalid position specification is set to 0. GOTOXY uses the other specified position if it is valid. If both *integer-expression1* and *integer-expression2* are invalid, two warnings are displayed and the cursor is set to position (0,0).

Example The following program prompts for a name and then concatenates the name with two other strings. The concatenated string is displayed in descending columns of the display. The FOR-loop causes the computer to pause long enough for the message to be viewed before it moves to the next column.

```
100 PROGRAM exgotoxy;
110 VAR strname,message:STRING;
120     col,pause:INTEGER;
130 BEGIN
140   message:='Here''s running Pascal';
150   WRITE('Enter your name: ') {$w-};
160   READLN(strname);
170   INSERT(strname,message,8);
180   FOR col:=30 DOWNTO 0 DO
190     BEGIN
200       GOTOXY(col,0);
210       WRITE(message);
220       FOR pause:=1 TO 15 DO
230         strname:=strname;
              (* occupy time for display *)
240       WRITELN;
250     END;(* col 30 to 0 *)
260 END. (* exgotoxy *)
```

HALT

Syntax "HALT"

Description The HALT procedure terminates program execution and causes the error message `Program halt` to be displayed. HALT should be used in exceptional cases, not terminate a program normally. Note that HALT cannot be used in an imperative.

Example The following program reads a real value and an integer from the keyboard. If the integer is less than zero or greater than 37, the program halts. Otherwise, the real value is multiplied by 10 raised to the power specified by the integer.

```
100 PROGRAM exhalt;
110 VAR intval:INTEGER;
120     realval:REAL;
130 BEGIN
140   WRITE('Enter value and power of ten : ')
       {$w-};
150   READLN(realval,intval) {$w+};
160   IF(intval<0) OR (intval>37)
170     THEN HALT
180     ELSE WRITELN('Value is:
         realval*PWROFTEN(intval);
190 END. (* exhalt *)
```

identifier

Syntax *letter* { *letter* ¦ *digit* }

Description Identifiers are any words or names in Pascal. In TI-74 Pascal, pre-defined identifiers are listed in uppercase letters; user-defined identifiers are listed in lowercase letters. A maximum of eight characters in a user-defined identifier are retained by the interpreter.

Syntax "IF" *Boolean-expression* "THEN" *statement* ["ELSE" *statement*]

Description The IF statement determines which one of two options is selected. If *Boolean-expression* is TRUE, the *statement* following the reserved word THEN is executed. If *Boolean-expression* is not TRUE, the *statement* following THEN is ignored and the next *statement* is executed.

If the ELSE part is included and *Boolean-expression* is TRUE, the *statement* following the reserved word THEN is executed, the ELSE part is ignored. If *Boolean-expression* is not TRUE, the *statement* following THEN is ignored and *statement* following ELSE is executed.

Statement following THEN or ELSE can be a single *statement* or a group of statements enclosed by BEGIN and END into a single compound statement.

IF statements can be nested by including an IF statement after either the word THEN or ELSE. An ELSE that appears in a nested IF statement is matched with the closest IF that has not been matched or closed off by the reserved word END. For example, in the statements

```
IF FALSE<TRUE
  THEN IF a<b
    THEN WRITELN('false<true and a<b')
    ELSE WRITELN('false<true but a not<b');
```

the ELSE is matched with the IF statement that compares a and b. An ELSE statement can be matched with another IF statement by using the reserved words BEGIN and END around the THEN part of the statement. For example, in the statements

```
IF FALSE<TRUE
  THEN
    BEGIN
      IF a<b
        THEN WRITELN('(false<true) and
        (a<b)')
    END (* no semicolon before an else *)
  ELSE WRITELN('false not less than true');
```

the ELSE is part of the IF statement that compares FALSE and TRUE.

Example The following program accepts an angle code and an angle measurement. If the angle code specifies degrees or grads, the angle measurement is converted to radians. The cosine of the angle is then displayed. The program repeats as long as the first character entered is a d, g, or r.

```
100 PROGRAM exit;
110 CONST pi=3.14159265359;
120 VAR rval:REAL;
130     ch:CHAR;
140 BEGIN
150   WRITE('Enter d, r, or g and angle: ')
       {$w-};
160   READLN(ch,rval);{$w+}
170   WHILE ch IN['d','D','r','R','g','G'] DO
180     BEGIN
190       IF(ch='d') OR (ch='D')
200         THEN rval:=rval*pi/180
210         ELSE
220           IF(ch='g') OR (ch='G')
230             THEN rval:=rval*pi/200.0;
240       WRITELN('cos: ',COS(rval);
250       READLN(ch,rval);
260     END;(* while valid argument *)
270 END. (* exit *)
```

IN

Syntax

simple-expression “IN” “[“ *ordinal-subrange* { “,” *ordinal-subrange*} “]”

where *ordinal-subrange* is

ordinal-type
¦
ordinal-type “..” *ordinal-type*

Description

The relational operator IN is used to determine set membership. The right operand can be an INTEGER, BOOLEAN, or CHAR set. If the operand on the left is a member of the operand on the right, IN returns a value of TRUE; otherwise, IN returns a value of FALSE.

The set defined in the right operand is a subset of any valid data type. If a subrange is an element of the set, the first value specified in the subrange must be less than or equal to the last value specified. For example, the set 'a'..'z' is a valid subrange of the type CHAR.

Examples

`4 IN[3,7,5,8]` is FALSE
`'d' IN['a'..'z']` is TRUE
`alphanum IN['A'..'Z','a'..'z','0'..'9']` is TRUE if the CHAR variable `alphanum` is an alphabetic or numeric character

INPUT

Syntax The file INPUT is always open and does not need to be declared.

Description The reserved word INPUT is a predefined file of type INTERACTIVE. The file INPUT reads characters entered from the keyboard and displays them.

INSERT

Syntax "INSERT" "(" *string-expression* "," *string-variable* "," *integer-expression* ")"

Description The INSERT procedure inserts a string into another string. *String-expression* is inserted into *string-variable*, starting at the index position in *string-variable* specified by *integer-expression*. The new length of *string-variable* is its previous length plus the length of *string-expression*. If inserting *string-expression* into *string-variable* would cause *string-variable* to be longer than its declared maximum length (default length if one is not specified), no characters are inserted.

Example The following program reads a name and a box number. The box number is inserted into an address string. The name and the address are then printed.

```
100 PROGRAM exinsert;
110 VAR st1,st2,address,message:STRING;
120     col,pause:INTEGER;
130     filepr:TEXT;
140 BEGIN
150   REWRITE(fiIepr,'20');
160   message:='P.O. Box #'
170   WRITE('Enter name: ') ($w-};
180   READLN(st1);
190   WRITE('Enter box number : ');{$w-}
200   READLN(address);
210   INSERT(address,message,ll);
220   WRITE(filepr,stl,' ');
230   WRITELN(filepr,message);
240   CLOSE(filepr,LOCK);
250 END. (* exinsert *)
```

INTEGER

Syntax “INTEGER”

Description The predefined ordinal data type INTEGER is used to represent the counting numbers, their negatives, and zero. The largest integer allowed is called MAXINT and has a value of 32767. The smallest integer that can be entered is -32767 (although the number -32768 can be used in calculations).

The arithmetic operators that produce an integer result are +, -, *, DIV, and MOD. The seven relational operators (=, <>, <, <=, >, >=, and IN) can be used with INTEGER data to obtain Boolean results. The following functions can be used to obtain an integer value.

ABS
SQR
TRUNC
ROUND
MEMAVAIL
SCAN
SIZEOF
LENGTH
POS
ORD
IORESULT

INTERACTIVE

Syntax “INTERACTIVE”'

Description INTERACTIVE is a predefined file type. The three INTERACTIVE file types are INPUT, OUTPUT, and KEYBOARD, which are always open. An INTERACTIVE is similar to a TEXT file; however, the EOF and EOLN functions are used differently with INTERACTIVE files than with TEXT files.

interpreter options

Syntax

"(*" "$" "i" ¦ "w" "+" ¦ "-" [*remarks*] "*)"
¦
"{" "$" "i" ¦ "w" "+" ¦ "-" [*remarks*] "}"

where *remarks* is a sequence of characters that does not contain closing delimiter of comment

Description

The options that you can specify to the interpreter as it executes a program enable you to:

- have the computer wait or not wait after characters are sent to the display (w).
- have the computer check or not check input/output operations (i).

An option is specified in a comment anywhere in a program after the first reserved word BEGIN in the program body. As the computer scans the program lines during execution, an interpreter option is turned on or off as specified only if the statement containing the comment is executed. Only one option can be included per comment.

To specify an option, place a $ immediately after the opening delimiter, (* or {, of a comment, followed by the letter w (wait) or i (input/output check). A plus sign (+) written after the letter causes the interpreter to turn on the option; a negative sign (-) written after the letter causes the interpreter to turn off the option.

After the RUN command is entered and the interpreter encounters the first reserved word BEGIN in the program body, the interpreter sets the default options as shown on the next page.

Letter	Default	Option
w	+	the interpreter suspends execution of a program when the program writes characters to the display. This delay gives you time to view the display. When either the CLR or ENTER key is pressed, program execution is resumed.
i	+	the interpreter checks input/output operations. See IORESULT in chapter 8 of the User's Guide for information on how input/output operations can be checked in a program.

When a breakpoint occurs, all current option settings are saved. During the break, each interpreter option is changed to its default status. When program execution is resumed, each option is reset to the setting that was in effect when the breakpoint occurred.

After a program completes execution, each interpreter option is set to its default status.

IORESULT

Syntax "IORESULT"

Description The IORESULT function returns the result of the last input/output operation performed. This function can be used to check I/O operations within a program after the interpreter option i (input/output check) has been turned off to keep the Pascal system from checking input/output operations.

A zero value returned by IORESULT means that the last I/O operation was performed with no errors. The other values returned by IORESULT are listed in the table on the next page and in more detail in appendix I.

Code	Definition
1	DEVICE/FILE OPTIONS ERROR
2	ERROR IN ATTRIBUTES
3	FILE NOT FOUND
4	DEVICE/FILE NOT OPEN
5	DEVICE/FILE ALREADY OPEN
6	DEVICE ERROR
7	END OF FILE
8	DATA/FILE TOO LONG
9	WRITE PROTECT ERROR
10	NOT REQUESTING SERVICE
11	DIRECTORY FULL
12	BUFFER SIZE ERROR
13	UNSUPPORTED COMMAND
14	DEVICE/FILENOT OPENED FOR OUTPUT
15	DEVICE/FILENOT OPENED FOR INPUT
16	CHECKSUM ERROR
20	OUTPUT MODE NOT SUPPORTED
2l	INPUT MODE NOT SUPPORTED
24	VERIFYERROR
25	LOW BATTERIES IN PERIPHERAL
26	UNINITIALIZED MEDIUM
32	MEDIUM FULL
80	SYNTAX ERROR IN DATA/DATA OVERRUN
81	PARITY ERROR
82	FRAMING ERROR
83	FRAMING AND PARITY ERRORS
255	TIME-OUT ERROR

Example The following program reads a filename and prompts for a medium to be loaded into a mass-storage device. After the ENTER key is pressed, the system I/O check is turned off and the program attempts to open the specified file. IORESULT is used to determine whether the file was opened. If an irrecoverable error occurs, the program halts. If IORESULT returns a code of 3, 7, or 9 the program repeats until an error code of 0 is returned or an irrecoverable error occurs. After the file is open, the data is read and displayed.

```
PROGRAM exiores;
VAR strname,filename:STRING;
    file1:TEXT;
    iocheck:INTEGER;
BEGIN
  REPEAT
    WRITE('Enter filename: ') {$w-};
    READLN(strname) {$w+};
    WRITELN('Load cartridge into drive');
     {$i-  turn off I/O check}
    fiIename:=CONCAT('1.',strname);
    RESET(file1,filename);
    iocheck:=IORESULT;
    IF NOT(iocheck IN[0,3,7,9]) THEN HALT;
    CASE iocheck OF
      3:WRITELN('Can''t find file-check cartridge');
      7:WRITELN('Empty file-check cartridge');
      9:WRITELN('Write protected-check cartridge');
    END;
  UNTIL iocheck=0;{$i+  turn on I/ O check}
  WHILE NOT EOF (file1) DO
    BEGIN
      READLN(file1,strname);
      WRITELN(strname);
    END;(* while not end of file *)
END. (* exiores *)
```

KEYBOARD

Syntax The file KEYBOARD is always open and does not need to declared.

Description The reserved word KEYBOARD is a predefined file of type INTERACTIVE. The file KEYBOARD is similar to the file INPUT except that the characters entered from the keyboard are not displayed (or sent to the fiie OUTPUT). The file KEYBOARD is useful when the input should not be displayed

LABEL

Syntax “LABEL” *integer* { “,” *integer*} “:”

Description The LABEL declaration is used to declare an integer that can be used as a label. A label must be an integer in the range 0..9999, followed by a colon and placed at the beginning of a statement. Any declared label must appear before one (and only one) statement in the program. A label must be in the same block as the GOTO statement that references it.

In a block, a LABEL declaration must precede any other declarations.

LENGTH

Syntax ”LENGTH” “(“ *string-expression* “)”

Description The LENGTH function returns the length of *string-expression*. The returned value is an INTEGER type and is equal to the number of characters in *string-expression*.

Example The following program reads a string from the keyboard, replaces any blanks with underlines, and displays the string.

```
100 PROGRAM exlength ;
110 VAR strval:STRING ;
120     count:INTEGER;
130 BEGIN
140   WRITE('Enter string: ') {$w-};
150   READLN(strval) {$w+};
160   FOR count:=1 to LENGTH(strval)
170     IF strval[count]=' '
180       THEN strval[count]:='_';
190   WRITELN(strval);
200 END. (* exlength *)
```

LIST

Syntax “LIST” [*line-number-list*]

|

“LIST” “,” *device* “.” *filename* [“,” *line-number-list*]

Description The LIST command lists the lines of the program currently in memory. If no line numbers are specified, the entire program is listed. Otherwise, only the lines specified are listed. *The line-number-list* may be any one or any combination of the following (separated by commas).

Line-number-list	**Action**
a single line number	lists that line
line number-	lists that line and all succeeding lines
-line number	lists that line and all preceding lines
line number-line number	lists that inclusive range of lines

When *device.filename* is given, the lines are listed to the specified device. For a listing to a printer, *filename* specifies the options required for the printer.

If *device.filename* is omitted, the lines are displayed. During a listing to the display, the lines may be edited. Pressing the t key terminates a listing to the display.

Example

```
LIST
```

displays the entire program currently in memory.

```
LIST 150-190,250-280,350-
```

displays lines 150 through 190, lines 250 through 280, and lines 350 through the end of the program.

```
LIST '10'
```

lists the entire program currently in memory to device 10.

```
LIST '50.R=L',-150
```

lists all lines up to and including line 150 to peripheral device 50.

LN

Syntax “LN” “(” *integer-expression* ¦ *real-expression* “)”

Description The LN function returns the natural logarithm of *integer-expression* or *real-expression*. LN returns a real value that the logarithm of *integer-expression* or *real-expression* to the base *e*, where *e* equals 2.71828182846. *Integer-expression* or *real|-expression* must be a value greater than zero, or an error occurs. The LN function is the inverse of the EXP function.

Example The following program reads two numbers from the keyboard and finds the logarithm of the first number to the base specified by the second number. The program repeats until a negative number or zero is entered.

```
100 PROGRAM exln;
110 VAR rval,base:REAL;
120 BEGIN
130   WRITELN('To stop enter neg # or zero');
140   WRITE('Enter value and base: ') {$w-};
150   READLN(rval,base) {$w+};
160   WHILE rval>0.0 DO
170     BEGIN
180       WRITELN('Log: ',rval,' to base ',base,': ',
           LN(rval)/LN(base));
190       WRITE(' Enter value and base: ') {$w-};
200       READLN(rval,base) {$w+};
210     END;(* while argument valid *)
220 END. (* exln *)
```

LOG

Syntax “LOG” “(” *integer-expression* ¦ *real-expression* “)”

Description The LOG function returns the natural logarithm of *integer-expression* or *real-expression*. LOG returns a real value that the logarithm of *integer-expression* or *real-expression* to the base 10. *Integer-expression* or *rea|-expression* must be a value greater than zero, or an error occurs. The LN function is the inverse of the EXP function.

Example The following program displays the common logarithm of numbers entered from the keyboard until a negative number or zero is entered.

```
100 PROGRAM exlog;
110 VAR rval:REAL;
120 BEGIN
130   WRITE('Enter value (zero to stop): ') {$w-};
150   READLN(rval) {$w+};
160   WHILE rval>0.0 DO
170     BEGIN
180       WRITELN('LOG: of',rval,': ',LOG (rval);
190       WRITE('Enter value (zero to stop): ') {$w-};
200       READLN(rval) {$w+};
210     END;(* while argument valid *)
220 END. (* exlog *)
```

MAXINT

Syntax "MAXINT"

Description MAXINT is a predefined integer constant identifier whose value is 32767.

MEMAVAIL

Syntax “MEMAVAIL”

Description The MEMAVAIL function returns an integer value equal to the number of bytes of memory not being used by the program currently in memory.

Note: In versions of Pascal written for 16-bit processors, MEMAVAIL returns the number of unallocated 16-bit words in memory.

Example The following program displays the number of unused bytes in memory.

```
100 PROGRAM exmemav;
110 BEGIN
120   WRITELN('Memory available is '
       MEMAVAIL,' bytes');
130 END. (* exmemav *)
```

MOD

Syntax *integer-expression* "MOD" *integer-expression*

Description The operator MOD computes the quotient of the left divided by the right operand and returns the remainder. The result is an INTEGER value. If the second operand has a value of zero, an error occurs.

The MOD operator is on the same precedence level as the operators AND, *, /, and DIV.

Example

```
100 PROGRAM exmod;
110 BEGIN
120   WRITELN('8 MOD 3 = ',8 MOD 3);
130 END. (* exmod *)
```

Output:

```
8 MOD 3 = 2
```

MOVELEFT

Syntax

"MOVELEFT" "(" *variable1* "," *variable2* "," *integer-expression* ")"

Description

The MOVELEFT procedure moves a specified number of bytes to another part of memory. MOVELEFT moves the data, one byte at a time, beginning with the byte stored in *variable1* .The data is moved to the byte specified by *variable2*. MOVELEFT continues to move the data in each successive byte after *variable1* to each successive byte after *variable2* until *integer-expression* bytes have been moved. The variables can be any type except a file type.

Note that when a string variable is used in MOVELEFT, the variable should be indexed.

Warning: Use care when moving data with MOVELEFT. If system data is written over, the computer may have to be reset to continue operation.

Example

The following program moves part of `st2` into `st1` and then displays the modified string.

```
100 PROGRAM exmove1;
110 VAR st1,st2:STRING;
120 BEGIN
130   st1:='move from the left,
       move to the right';
140   st2:='move your data, byte by byte';
150   MOVELEFT(st2[17],stl[26],12);
160   WRITELN(stl);
170 END. (* exmovel *)
```

Output:

```
move from the left, move byte by byte
```

MOVERIGHT

Syntax

"MOVERIGHT" "(" *variable1* "," *variable2* "," *integer-expression* ")"

Description

The MOVERIGHT procedure moves a specified number of bytes to another part of memory. MOVERIGHT moves the data, one byte at a time, beginning with the byte that is at *variable1* plus *integer-expression* minus 1. The data is moved to the byte specified by *variable2* plus *integer-expression* minus one. MOVERIGHT continues to move the data in each preceding byte before *variable1* plus *integer-expression* minus 1 to each preceding byte before *variable2* plus *integer-expression* minus one until *integer-expression* bytes have been moved.

Note that when a string variable is used in MOVERIGHT, variable should be indexed.

Warning: Use care when moving data with MOVERIGHT. If system data is written over, the computer may have to be reset to continue operation.

Example

The following program uses MOVERIGHT to move data within string `s`. Note that the contents of some bytes are modified before they are moved.

```
100 PR0GRAM exmover;
110 VAR s:STRING;
120 BEGIN
130   s:='move from the right, move to the left,
       byte by byte';
140   MOVERIGHT(s[34],s[20],18);
150   WRITELN(s);
160 END. (* exmover *)
```

Output:

```
move from the rightbyte, byte by byte, byte
byte
```

NEW

Syntax “NEW” [“ALL”]

Description The NEW command prepares the computer for a new program by deleting the program and variables currently in memory. All open files are closed. Each interpreter option is set to its default status as shown below.

w is set to +	data sent to the display remains there until the ENTER or CLR key is pressed.
i is set to +	input/output operations are automatically checked.
a is set to -	no beep occurs when the computer is accepting input.

The NEW ALL command clears user-assigned strings and all display indicators except RAD in addition to the initialization process performed by NEW.

Example

```
NEW ALL
```

clears all of memory and prepares the computer for a new program.

NOT

Syntax “NOT” *Boolean-expression*

Description The Boolean operator NOT is a unary operator that performs the negation of Boolean-expression. The order of for the logical operators from highest to lowest is NOT, AND, and OR. The operation NOT is performed before AND, and the arithmetic operators *, /, DIV, and MOD, before the operators +, -, and OR, and before the relational operators =, <>, <, <=, >, >=.

The following table gives the results of performing a NOT on a Boolean expression.

Example

Boolean-expression	`NOT` *Boolean-expression*
TRUE	FALSE
FALSE	TRUE

NUMBER

Syntax “NUMBER” ¦ “NUM” [*initial-line*] [“,” *increment*]

Description The NUMBER (or NUM) command generates sequenced line numbers. These line numbers are displayed with a trailing space for convenience when program statements are entered. Only the statement has to be typed. After ENTER is pressed, the program line is stored in memory and the next line number is displayed.

NUMBER starts numbering at 100 and increases in increments of 10. Optionally, the *initial-line* and *increment* can be specified to indicate where the numbering is to begin and what increment is used.

If a line specified by NUMBER already exists, that line is displayed and may then be replaced or changed using the edit functions. If the line number is altered, the sequence of generated line numbers continues from the new line number.

Note that if a line already exists with a line number not in the sequence being used by NUMBER, that line is not displayed by NUMBER.

To terminate the numbering process, press ENTER when a line is displayed with no statements on it, or press BREAK when any line is displayed.

Example

```
NUM
```

causes the computer to generate line numbers starting at 100 and use an increment of 10.

```
NUM 200,20
```

causes the computer to generate line numbers starting at 200 and use an increment of 20.

ODD

Syntax "ODD" "(" *integer-expression* ")"

Description The ODD function returns a Boolean value of TRUE if *integer-expression* is odd or a Boolean value of FALSE if *integer-expression* is even.

Example The following program displays the characters corresponding to the odd-numbered codes in appendix G.

```
100 PROGRAM exodd;
110 VAR count: INTEGER;
120 BEGIN
130   FOR count:=32 TO 255 DO
140     IF ODD(count)
150       THEN WRITELN(count,' ',
           CHR(count));
160 END. (" exodd *)
```

OF

Syntax Refer to ARRAY, FILE, and CASE

Description The reserved word OF is used in the CASE statement and in ARRAY and FILE declarations. A semicolon (;) should not appear immediately after the reserved word OF.

OLD

Syntax “OLD” “'” device ”.” filename “'”

Description The OLD command loads a program from an external device into memory. OLD closes all open files and, if it finds *filename*, removes the program currently in memory and loads *filename*. A Pascal program can be stored on *device.filename* with the SAVE command.

Device.filename identifies the device where the program is stored and the name of the file. *Device* is the number associated with the peripheral device and can be from 1 through 255. *Filename* identifies the particular file. Refer to the peripheral manuals for the device code for each peripheral and for specific information about the form of *filename*.

Warning: If filename specifies a data file rather than a program file, this data may write over system data. The RESET key will then have to be pressed to restore the system data.

Example

```
OLD '1.prog10'
```

Loads the program `prog10` on device 2 into memory.

OR

Syntax *Boolean-expression1* "OR" *Boolean-expression2*

Description The operator OR performs the logical disjunction of two Boolean expressions. The order of precedence for logical operators from highest to lowest is NOT, AND, and OR. The operator OR is performed on the same level as the operators +, and -. OR is performed after the operators *, /, DIV, MOD, and AND, and before the relational operators =, <>, <, <=, >,>= , and IN.

The following table gives the results of performing an OR on all possible combinations of Boolean expressions.

Example

Booexp1/Booexp2	**TRUE**	**FALSE**
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE

ORD

Syntax “ORD” “(” *ordinal-expression* “)”

Description The ORD function returns the ordinal number of *ordinal-expression* in the set of values valid for that type. If *ordinal-*expression is a CHAR type, the value returned is the code of the character. Refer to appendix G for a list of the ASCII codes. If *ordinal-expression* is an INTEGER type, value returned is the value of *ordinal-expression*. The ordinal value of FALSE is 0;the ordinal value of TRUE is 1.

Example The following program reads a string from the keyboard and changes any lowercase letters to uppercase. The string is displayed.

```
100 PROGRAM exord;
110 VAR strval:STRING;
120     count:INTEGER;
130 BEGIN
140   WRITE('Enter string :') {$w-};
150   READLN(strval) {$w+};
160   FOR count:=1 TO LENGTH(strval) DO
170     IF(ORD(strval[count])>=ORD('a')
175      AND(ORD(strval[count])<=ORD('z')
180     THEN strval[count]:=
         CHR(ORD(strval[count])-32);
190   WRITELN(strval);
200 END. (* exord *)
```

ordinal type

Syntax

INTEGER
¦
BOOLEAN
¦
CHAR
¦
subrange of INTEGER, BOOLEAN, or CHAR

Description

An ordinal type is a set of finite values that can be ordered. In TI-74 Pascal, the ordinal types are INTEGER, CHAR, and BOOLEAN and any subrange of them.

OUTPUT

Syntax The file OUTPUT is always open and does not need to be declared.

Description The reserved word OUTPUT is a predefined file of type INTERACTVE. The OUTPUT file writes its data to the display. The file OUTPUT is assumed when a file-identifier is not used in an output statement.

PACKED

Syntax Refer to ARRAY

Description The reserved word PACKED is used to store data in an array in the least amount of space necessary. The TI-74 uses bytes to store values and therefore data cannot be stored in any less space than is normally used. Whole array operations can be performed only on arrays of type PACKED.

PAGE

Syntax “PAGE” [“(” *file-identifier* “)”]

Description The PAGE procedure sends a form feed (page advance) character to the file specified by *file-identifier*, where *file-identifier* is the identifier of an open text file. When *file-identifier* is omitted, the file OUTPUT is assumed and the display is cleared and its cursor moved to column 1.

If no end-of-line marker is written to a printer before a PAGE procedure is encountered, PAGE writes the end-of-line marker before sending the form feed character.

Example The following program reads a specified number of topics and prints each topic at the top of a page along with a page number.

```
100 PROGRAM expage;
110 VAR filepr:TEXT;
120     count,index:INTEGER;
130     strval:STRING;
140 BEGIN
150   REWRITE(filepr,'20');
160   WRITE('Enter number of topics: ') {$w-};
170   READLN(count) {$w+};
180   WRITELN('Enter topics to be discussed: ');
190   FOR index:=l TO count DO
200     BEGIN
210       READLN(strval);
220       PAGE(filepr);
230       WRITELN(fiLepr,'                     ',INDEX);
240       WRITELN(filepr,strval);
250     END;(* index for aLL pages *)
260 END. (* expage *)
```

Syntax “POS” “(”*string-expressionl* “,” *string-expression2* “)”

Description The POS function returns the position of the first occurrence of one string within another string. POS scans *string-expression2* to determine if *string-expressionl* can be found. POS returns an integer value that is the index position in *string-expression2* where *string-expression2* first matched *string-expressionl*. If no match is found, POS returns a zero.

Example The following program reads a number. The number is inserted into the string `st2` at the location (plus 1) of the # sign in the string. The strings `st1` and `st2` are printed at the start of a new page. The order number is then deleted from the string. The program repeats if the Y or y key is pressed.

```
PROGRAM expos;
VAR st1:STRING[15];
    st2:STRING;
    stnum:STRING[20];
    filepr:TEXT;
    ch :CHAR;
BEGIN
  stl:='Dear Customer, ';
  st2:='Your order # is processed';
  REWRITE(filepr,'20');
  REPEAT
    WRITE('Enter order #: ') {$w-};
    READLN(stnum) {$w+};
    INSERT(stnum,st2,POS('#',st2)+1);
    PAGE(filepr);
    WRITELN(filepr,st1);
    WRITELN(filepr,st2);
    DELETE(st2,POS('#',st2)+1,LENGTH(stnum));
    WRITE('Continue (Y or N) ') {$w-};
    READLN(ch) {$w+};
  UNTIL NOT((ch='Y') OR (ch='y'));
END. (* expos *)
```

PRED

Syntax “PRED” “(” *ordinal-expression* “)”

Description The PRED function returns the value that precedes *ordinal-expression* in the set of values to which *ordinal-expression* belongs. The value returned is the same type as *ordinal-expression*. If *ordinal-expression* is the first value in its set, an error occurs when PRED is executed.

Example

```
100 PROGRAM expred;
110 VAR int1,int2: INTEGER;
120 BEGIN
130   WRITE('Enter range for values: ');
       {$w-}
140   READLN(int1,int2);{$w+}
150   WRITE(int1,' to ', int2,' is ',
       int1,' through ', PRED(int2));
160 END. (* expred *)
```

Input:

```
1 13
```

Output:

```
1 to 13 is 1 through 12
```

PROCEDURE

Syntax “PROCEDURE” *identifier* [“(” *parameter-list* “)”] “;” block “;”

where parameter-list is

[VAR] *identifier* { “,” *identifier*} “:” *type-identifier* { “;”
[VAR] *identifier* { “,” *identifier*} “:” *type-identifier*}

Description A procedure is a block within a program that is declared in a PROCEDURE declaration with an identifier so that the block can be executed when its identifier is used as a statement. When a procedure identifier appears in a program as a statement, the statement is known as a procedure call. When this call is executed, the statements defined as the specified procedure are executed.

When a procedure is called, an unallocated part of memory is used for its local variables. After the procedure terminates, the memory is released. Any identifiers defined within a procedure are local to that procedure and are undefined outside that procedure. If a local variable has the same identifier as a global variable, the local variable supersedes the global variable.

A PROCEDURE declaration may also contain parameters to transfer data between a procedure and the block in which it is defined. The parameters can be value parameters or VAR parameters.

A parameter that is a value parameter is a local variable within a procedure. When the procedure is called, the values of the parameters in the procedure call are evaluated and assigned to the corresponding value parameters in the procedure. Any change made to a value parameter is not made to the corresponding parameter used in the procedure call.

A parameter that is a VAR parameter uses the storage location of its corresponding formal parameter in the procedure call. Any change made to a VAR parameter is also made to its corresponding parameter in the procedure call because they use the same memory locations.

A procedure can be recursive.

PROGRAM

Syntax “PROGRAM” *identifier* [“(” *file-identifier* { “,” file-identifier } “)”] “;” *block*

Description A program consists of a program heading, a declarations section, and a statements section. The identifier used in program heading may not be used as an identifier in the program. An identifier listed in the program heading is checked for syntax, but is otherwise ignored.

Every program must have a program heading and a body. The program body is the group of statements by the reserved words BEGIN and END.

PWROFTEN

Syntax “PWROFTEN” “(” *integer-expression* “)”

Description The PWROFTEN function returns the value of ten raised to the power specified by *integer-expression*. *Integer-expression* must be an integer from 0 through 37. PWROFTEN returns a REAL value.

Example The following program reads an angle measured in degrees, converts it to radians, and displays its tangent if it exists. The program repeats as long as the magnitude of the angle is less than $90*10^{10}$.

```
100 PROGRAM expwrten;
110 CONST pi=3.1415926359;
120 VAR degval,radval,max:REAL;
130 BEGIN
140   max:=90*PWROFTEN(10);
150   WRITE('Enter degrees : ') {$w-};
160   READLN(degval) {$w+};
170   WHILE ABS(degval)<max DO
180     BEGIN
190       radval:=degval*pi/180;
200       IF COS(radval)=0.0
210         THEN
220           WRITELN('Tan of ',degval,'not defined')
230         ELSE
240           WRITELN('Tan of ', degval,' deg: ',
               SIN(radval)/COS(radval);
250       WRITE('deg : ') {$w-};
260       READLN(degval) {$w+};
270     END;(* while valid angle *)
280 END. (* expwrten *)
```

READ

Syntax “READ” “(” [*file-identifier* “,”] *variable* { “,” *variable* } “)”

Description The READ statement assigns values to its list of variables. When a READ statement is executed, the computer suspends program execution and reads values until it encounters an end-of-line marker. The values read are assigned from left right to the variables listed in the READ statement.

If more values are read than there are variables in the READ statement, the values are retained in an input buffer for the next input statement.

If the next input statement is another READ statement, the variables in this READ statement are assigned values from the extra values retained in the input buffer. If any values still remain unassigned, they are retained until the next statement is encountered.

If the next input statement is a READLN statement, the variables in the READLN statement are assigned values from the extra values retained in the input buffer. If any values still remain unassigned, they are discarded.

Variable cannot be of type BOOLEAN. A *variable* of type PACKED ARRAY OF CHAR must be indexed for each of it dimensions. The value assigned to a variable must be a valid data type for that variable.

File-identifier is the identifier of a text file opened by RESET. If *file-identifier* is omitted, the computer reads from the INPUT file.

The function EOF(*file-identifier*) must be false before a READ(*file-identifier*) statement is executed. Otherwise, an error occurs and the program is terminated.

Refer to the READLN statement for information on how an input statement reads INTEGER, REAL, CHAR, and STRING data.

Example

The following program reads three characters, two integers, and a real value as shown below. Note that when `ch3` is read, the EOLN marker is the next character after the character `'b'`. Thus, `ch3` is assigned the blank character.

```
100 PROGRAM exread;
110 VAR ch1,ch2,ch3:CHAR;
120     int1,int2:INTEGER ;
130     realval:REAL ;
140 BEGIN
150   READ(ch1,ch2);
160   READ(ch3,int1);
170   READ(realval,int2);
180   WRITELN(ch1:5,ch2:5,ch3:5);
190   WRITELN(int1:10,realval:10,int2:10);
200 END. (* exread *)
```

Input lines:

```
ab
500 123.5
700
```

Output:

```
    a    b
       500     123.5       700
```

READLN

Syntax "READLN" ["(" [*file-identifier* ","] *variable* { "," *variable*} ")"]

Description The READLN statement assigns values to its list of variables. When a READLN statement is executed, the computer suspends program execution and reads values to assign to its variables from left to right. After the last variable has been assigned a value, the READLN moves past the end-of-line marker to the first character in the next line.

If no variables are included in a READLN statement, the READLN moves past the end-of-line marker to the first character in the next line.

When the computer is reading data to assign to an INTEGER or a REAL type variable, all leading blanks and end-of markers are skipped until a nonblank is reached. The nonblank character must be a sign or a digit.

All characters after the nonblank character are read until a nonnumeric character is reached. For an INTEGER type, the nonnumeric character causes the computer to stop reading characters for that integer variable. For a REAL type, all characters after the nonblank are read until a nonnumeric character is reached that is not a valid character for a number expressed in either decimal or scientific notation. A value with no decimal point can be entered and the system will add the decimal point followed by a zero.

For a STRING type, all characters are read up to the end line marker (ENTER) or the end-of-file marker. Note that only one string may be read per READLN statement. If a READ statement reads a string, all characters up to the end-of-line marker are assigned to the string. If the next READLN statement reads a character, a blank is assigned the CHAR type variable. If the next READ or READLN statement reads a string, the null string is assigned to the string variable.

For a CHAR type, the next character is stored in the *variable* and the pointer advances one column. If the character just read is the last one in the line, the pointer is then positioned to the end-of-line marker. If the end-of-line marker is read, a space is stored in the variable and the pointer moves to the first character in the next line.

Variable cannot be of type BOOLEAN. A variable of type PACKED ARRAYOF CHAR must be indexed for each of its dimensions. The value assigned to a variable must be a valid data type for that variable.

File-identifier must be the identifier of a file opened by RESET. If *file-identifier* is omitted, the computer reads from the INPUT file.

The function EOF(*file-identifier*) must be false before a READLN(*file-identifier*) statement is executed. Otherwise, an error occurs and the program is terminated.

If the EOLN function is true when a READLN statement is executed, the value assigned variable in the READLN statement is dependent upon the data type of variable. For INTEGER and REAL data, the end-of-line marker is skipped. For CHAR data, a blank is assigned to the character variable. For STRING data, the null string is assigned to the string variable.

Example

The following program uses READLN to read three characters, an integer, a real value, another integer, and a string. The input is then displayed. A string is read from the file "datafile" on device1 and displayed. (Note that this file must already exist.) If the data shown as input is entered, the data shown as output is displayed.

```
100 PROGRAM exreadln;
110 VAR ch1,ch2,ch3:CHAR;
120     int1,int2:INTEGER;
130     realval:REAL;
140     st:STRING;
150     file1:TEXT;
160 BEGIN
170   READLN(ch1,ch2);
180   READLN(ch3,int1);
190   READLN(realval,int2);
200   READLN(st);
210   WRITELN(ch1:5,ch2:5,ch3:5);
220   WRITELN(intl:10,realval:10,int2:10);
230   WRITELN(st);
240   WRITELN;
250   RESET(file1,'1.datafile');
260   READLN(file1,st);
270   WRITELN(st);
280   CLOSE(file1,LOCK);
290 END. (* exreadln *)
```

Input:

```
ab
c 500
123.5 700
This is the end

This string is the first record of the file
datafile
```

Output:

```
    a    b    c
       500     123.5       700
This is the end

This string is the first record of the file datafile
```

REAL

Syntax "REAL"

Description The predefined type REAL represents fractional numbers and numbers of very large or small magnitude. The magnitude of REAL values in the TI-74 can be as small as l.0E-128 or as large as 9.9999999999999E+127. A REAL data type is written in scientific notation if the item has 11 or more digits to the left of the decimal point.

The operators that can be used to obtain a REAL value are the operators +, -, *, and /. Note that / can be used with INTEGER type values to obtain a REAL result. The relational operators =, <>, <, <=, >, and >= can be used with REAL data to obtain a BOOLEAN result. The following functions can be used to obtain a REAL value.

ABS
ATAN
COS
EXP
LN
LOG
PWROFTEN
SIN
SQR
SQRT

recursion

Description

Recursion is the execution of a procedure or a function on itself. A routine cannot call itself indefinitely or an overflow condition occurs. A recursive routine must contain a method of termination. When the condition of termination is met, the recursive routine returns control to the point where the procedure or function was originally called.

A routine can return immediately from anywhere in its body by calling the procedure EXIT.

When it is necessary to reference a procedure or function before it can be defined, the FORWARD declaration must used.

Example

```
100 PROGRAM fibo;
110 VAR fibnum:INTEGER;
120 FUNCTION fib(count:INTEGER): INTEGER;
130   BEGIN
140   IF count>1
150     THEN fib:=fib(count-1)+fib(count-2)
160     ELSE IF count=1
170       THEN fib:=1
180       ELSE fib:=0;
190   END;(* fib *)
200 BEGIN
210   REPEAT
220     WRITE({$w-}'Enter integer (1-10): ');
230     READLN(fibnum);{$w+}
240   UNTIL fibnum IN[1..10];
250   WRITELN('Fibonacci # ',fibnum,' is ',
       fib(fibnum));
260 END.(* fibo *)
```

RENUMBER

Syntax "RENUMBER" ¦ "REN" [*initial-line*] ["," *increment*]

Description The RENUMBER (or REN) command renumbers the line numbers of the program currently in memory, beginning with line 100 and using an increment of 10. Optionally, *initial-line* number and *increment* can be specified to begin renumbering at the specified line number using the given increment.

If the values entered for *initial-line* and *increment* result in the creation of a line number larger than 32766 or less than or equal to 0, the error message `Illegal line number` is displayed and the line numbers are left unchanged.

Examples

```
REN
```

renumbers the line numbers of the program currently in memory. The first line is numbered 100 and the line number of each succeeding line is increased by 10.

```
REN ,5
```

renumbers the line numbers of the program currently in memory. The first line is numbered 100 and the line number of each succeeding line is increased by 5.

REPEAT

Syntax

"REPEAT" *statement* { ";" *statement* } "UNTIL" *Boolean-expression*

Description

The REPEAT statement executes statement until *Boolean-expression* becomes true. *Statement* can be a single or several statements. The reserved words BEGIN and END are not necessary to enclose multiple statements because words REPEAT and UNTIL delimit them.

After statement is executed, *Boolean-expression* is evaluated to determine whether it is true or false. As long as *Boolean-expression* is false, *statement* is repeated. When *Boolean-expression* becomes true, execution continues with the next statement.

A REPEAT-loop is always executed at least one time

Example The following program reads a balance and adds deposits and subtracts checks from it. When the balance is less than zero, an overdraft message is displayed and the program prompts for a `y` key to be pressed to continue adding and subtracting entries. If a key other than `y` is pressed, the program terminates through an EXIT statement.

```
PROGRAM exrepeat;
VAR count,index:INTEGER;
    balance,rval:REAL;
    ch:CHAR;
BEGIN
  WRITELN('Enter checks as negative #''s &');
  WRITELN(' deposits as positive #''s');
  WRITELN('Enter 0 to stop ');
  WRITE('Enter balance: ') {$w-};
  READLN(balance);
  REPEAT
    WRITE('Transactions: ') {$w-};
    READLN(rval) {$w+};
    balance:=balance+rval {$w-};
    WRITELN('Balance: ',balance) {$w+};
    IF balance<0.0
      THEN
        BEGIN
          WRITE('Overdraft, press y to continue') {$w-};
          READLN(KEYBOARD,ch) {$w+};
          IF NOT((ch='Y') OR (ch='y'))
            THEN EXIT(PROGRAM);
        END;(* balance negative *)
  UNTIL rval=0.0;
END. (* exrepeat *)
```

RESET

Syntax “RESET” “(” *file-identifier* “,” “'” *device* “.” *filename* “'” “)” “;”

|

“RESET” “(” *file-identifier* “,” “'” *string-expression* “'” “)” “;”

Description The RESET procedure opens an existing file for input. When RESET opens a file, *file-identifier* is associated with the file specified by *filename* on *device*. *File-identifier* is the identifier used in input and output statements to access the elements of *filename*. If the file specified by *filename* do not exist, an error occurs.

String-expression must be in the form '*device.filename*'. *Device* is the number associated with the peripheral device and can be from 1through 255. Refer to the peripheral manuals for the device number of a device and for specific information about the form of *filename*.

If *file-identifier* references a file that is already open, an error occurs. A file must be closed before the RESET command can be used with it.

Example

The following program reads a specified number of strings from the keyboard and writes them to a file. After the strings are written, the file is closed and reset to the beginning of the file. Each string written to the file is then read and printed enclosed in apostrophes.

```
PROGRAM exreset;
VAR file1,filepr:TEXT;
    codestr,strdata:STRING;
    index,count:INTEGER;
BEGIN
  REWRITE(file1,'110.data10');
  WRITE('Enter number of lines: ') {$w-};
  READLN(count);{$w+}
  FOR index:=1 TO count DO
    BEGIN
      WRITE('? ') {$w-};
      READLN(strdata);
      WRITELN(file1.strdata);
    END;(* {$w+} for index *)
  CLOSE(file1,LOCK);
  RESET(file1,'110.data10');
  REWRITE(filepr,'20');
  WHILE NOT EOF (file1) DO
    BEGIN
      READLN(file1,strdata);
      WRITELN(filepr,'''',strdata,'''');
    END;(* EOF test *)
  CLOSE(file1,LOCK);
  CLOSE(filepr,LOCK);
END. (* exreset *)
```

REWRITE

Syntax “REWRITE” “(” *file-identifier* “,” “'” *device* “.” *filename* “'” “)” “;”

|

“REWRITE” “(” *file-identifier* “,” *string-expression* “)” “;”

Description The REWRITE procedure opens a file for output. REWRITE associates *file-identifier* with the file called *filename* and sets EOF true for that file. If *filename* already exists on device, the file must be closed when the REWRITE is executed. The EOF marker is written at the beginning of the file and any data in the file is lost. If *filename* does not already exist, a file is created with the name *filename* on device.

String-expression must be in the form *device.filename*. *Device* is the number associated with the peripheral device and can be from 1 through 255. Refer to the peripheral manuals for the device number of a device and for specific information about the form of *filename*.

Example The following program prompts for the number of the device to which the output is written. The number of specified strings are read from the keyboard and printed, one to a page.

```
100 PROGRAM exrewrit;
110 VAR filepr:TEXT;
120     codestr,strdata:STRING;
130     index,count:INTEGER ;
140 BEGIN
150   WRITE('Enter device # of printer: ')
       {$w-};
160   READLN(codestr);
170   REWRITE(filepr,codestr);
180   WRITE('Enter number of lines: ');
190   READLN(count);
200   FOR index:=1 TO count DO
210     BEGIN
220       WRITE(': ');
230       READLN(strdata);
240       PAGE(filepr);
250       WRITELN(filepr,'Page',index);
260       WRITELN(filepr,strdata);
270     END;(* for index *)
280 END. (* exrewrit *)
```

ROUND

Syntax “ROUND” “(” *real-expression* “)”

Description The ROUND function rounds *real-expression* to the nearest integer value. The value returned by ROUND is an integer. If the fractional part of *real-expression* is 0.5 or larger, the result is rounded up. Otherwise, *real-expression* is truncated.

Example The following program reads a dollar amount and the number of parts to divide into the amount. The result is rounded to the nearest cent and displayed.

```
PROGRAM exround;
VAR amount:REAL;
    group:INTEGER;
BEGIN
  WRITE('Enter total amount: ') {$w-};
  READLN(amount);
  WRITE('Enter number in group: ');
  READLN(group) {$w+};
  amount:=ROUND(amount/group*100.0)/
   100.0;
  WRITELN ('Each share is $',amount);
END. (* exround *)
```

Input:

```
345.62 5
```

Output:

```
Each share is $69.12
```

RUN

Syntax “RUN” [“‘” *device* “.” *filename* “‘”]
¦
“RUN” [string-expression]

Description The RUN command starts execution of a program at its lowest-numbered line. Optionally, RUN can load into and execute a program stored on *device* with the name *filename*.

String-expression must be in the form *device.filename*. *Device* is the number associated with the peripheral device and can be from 1 through 255. *Filename* identifies the of the program file. Refer to the peripheral manuals for the device number of a device and for specific information the form of *filename* .

After the RUN command has been entered but before the program actually begins execution, the following process takes place.

- Certain pre-run errors such as illegal nesting levels and matching parentheses are detected.
- All open files are closed.

Example

```
RUN
```

Instructs the computer to start executing the program currently in memory starting at the lowest-numbered line.

```
RUN '1.prog10'
```

instructs the computer to load into memory and execute program prog10 stored on device 1.

SAVE

Syntax “SAVE” “‘” *device* “.” filename “‘” [“,” “PROTECTED”]

Description The SAVE command enables you to copy the Pascal program currently in memory to an external device. By using the OLD or RUN command, you can later recall the program into memory.

Device.filename identifies the device where program *filename* is to be stored. *Device* is the number associated with the peripheral device and can be from 1 through 255. *Filename* identifies the file that is to contain the program. If PROTECTED is specified, the program in memory is left unprotected but the copy on the external storage device is saved in protected format. A protected program can be executed, but cannot be listed, edited, or saved.

Example

```
SAVE '1.prog10'
```

saves the program currently in memory to device 1 under the name prog10.

```
SAVE '3.progll',PROTECTED
```

saves the program currently in memory to device 3 under the name prog11. The program may be loaded into memory and run, but it may not be edited, listed, or resaved.

SCAN

Syntax

“SCAN” “(" *integer-expression* “,” “=” ¦ “<>” *character-expression* “,” *variable* “)”

Description

The SCAN function searches memory for an equal or unequal comparison to a character. *Integer-expression* specifies the number of bytes to search and. *variable* specifies where the memory search is to begin. The first byte of *variable* is the first byte searched for the indicated equality or inequality. *Variable* can be any type except a file type. If *integer-expression* is negative, the search is performed backwards in memory.

Character-expression must be preceded with either the equal sign(=) or the not equal sign (<>). An equal sign causes SCAN to search memory for the first byte that matches *character-expression*. The unequal sign causes SCAN to search for the first byte that does not match *character-expression*.

SCAN returns an integer value that is the number of bytes skipped during the search. If the first byte equals *character-expression* (or does not equal *character-expression*), SCAN returns a value of 0. If the second byte equals *character-expression* (or does not equal *character-expression*), SCAN returns a value of 1 and so on. If no equality (or inequality) was found, SCAN returns the value of *integer-expression*.

Note: If *variable* is a string variable, the string should be indexed. Otherwise, the SCAN function begins at the length byte of the string (rather than one of its characters).

Example The following program searches string `st` for the first character equal to “b” and the first character not equal to “z”. The results are shown below.

```
100 PROGRAM exscan;
110 VAR st:STRING;
120     ch:CHAR;
130 BEGIN
140   st:='SCAN returns the number of bytes
       skipped during a search';
150   ch:='b';
160   WRITELN(SCAN(LENGTH(st),=ch,st[1]);
170   WRITELN(SCAN(LENGTH(st),<>'z',st[1]);
180 END. (* exscan *)
```

Output:

```
20
0
```

SIN

Syntax “SIN” “(” *integer-expression* ¦ *real-expression* “)”

Description The SIN function returns the trigonometric sine of the qngle whose measurement in radians is *integer-expression* or *real-expression*. The absolute value of *integer-expression* or *real-expression* must be less than π/2 * 1010. SIN returns a real value.

Example The following program displays the trigonometric sine of a number entered from the keyboard. The program stops if a number with an absolute value greater than or equal to π/2 * 1010 is entered.

```
100 PROGRAM exsin;
110 CONST pi=3.14159265359;
120 VAR rval,max:REAL;
130 BEGIN
140   max:=pi/2*PWROFTEN(10);
150   WRITE('Enter angle in radians: ')
       {$w-}
160   READLN(rval) {$w+};
170   WHILE ABS(rval)<max DO
180     BEGIN
190       WRITELN('sin:',SIN(rval));
200       WRITE('rad:') {$w-};
210       READLN(rval){$w+};
220     END;(* while valid argument *)
230 END. (* exsin *)
```

SIZEOF

Syntax “SIZEOF” “(" *variable*: *type-identifier* “)”

Description The SIZEOF function returns the number of bytes a variable takes in memory. Variable can be any type variable except file. SIZEOF returns an integer value that is the number of bytes the specified variable occupies in memory or the number of bytes any variable of the specified datatype occupies. For a STRING item, the number of bytes returned is the maximum length of the string plus one for the length byte.

Example The following program reads a string and displays the number of bytes occupied by an INTEGER, REAL, or CHAR data type and the entered string.

```
100 PROGRAM exsizeof;
110 VAR st:STRING;
120 BEGIN
130   WRITE('Enter string: ') {$w-};
140   READLN(st) {$w+};
150   WRITELN('Number of bytes for each
       data type:');
160   WRITELN('INTEGER: ',SIZEOF(INTEGER);
170   WRITELN('REAL: ',SIZEOF(REAL);
180   WRITELN('CHAR: ',SIZEOF(CHAR);
190   WRITELN('String entered: ',SIZEOF(st));
200 END. (* exsizeof *)
```

Input:

```
This string is 33 characters long
```

Output:

```
Number of bytes for each data type:
INTEGER: 2
REAL: 8
CHAR: 1
String entered: 81
```

SQR

Syntax “SQR” “(” *integer-expression* ¦ *real-expression* “)”

Description The SQR function returns the square of *integer-expression* or *real-expression*. The value returned by SQR is of the same type as the expression.

Example The following program reads three integers entered in ascending order and ascertains if the three numbers form a right triangle. An appropriate message is then displayed. program is repeated until a zero is entered as the first integer.

```
100 PROGRAM exsqr;
110 VAR side1,side2,side3:INTEGER;
120 BEGIN
130   WRITELN('To stop enter 0');
140   WRITE('Enter 3 integers: ') {$w-};
150   READLN(side1,side2,side3) {$w+};
160   WHILE side1<>0 DO
170     BEGIN
180       IF SQR(side3)=SQR(side1)+SQR(side2)
190       THEN WRITELN('form right
           triangle')
200       ELSE WRITELN('do not form right
           triangle');
210       WRITE(': ') {$w-};
220       READLN(side1,side2,side3) {$w+};
230     END;(0 side1 = 0 *)
240 END. (* exsqr *)
```

Input:

```
6
8
10
```

Output:

```
form right triangle
```

SQRT

Syntax “SQRT” “(” *integer-expression* ¦ *real-expression* “)”

Description The SQRT function returns the square root of *integer-expression* or *real-expression*. *Integer-expression* or *real-expression* must be a positive number. The value returned by SQRT is a real value.

Example The following program computes and displays the real roots of a quadratic equation. If the roots are imaginary, an appropriate message is displayed. The program repeats until the number entered for the coefficient for the quadratic term is zero.

```
100 PROGRAM exsqrt;
110 VAR a,b,c,d:REAL;
120     root1,root2:REAL;
130 BEGIN
140   WRITELN('Enter 0 to stop');
150   WRITE('Enter a,b,c coefficients: ')
       {$w-};
160   READLN(a,b,c) {$w+};
170   WHILE a<>0.0 DO
180     BEGIN
190       d:=SQR(b)-4.0*a*c;
200       IF d>=0.0
210         THEN
220           BEGIN
230             root1:=(-b)+SQRT(d))/(2.0*a);
240             root2:=(-b)-SQRT(d))/(2.0*a);
250             WRITELN('Roots: ',root1:12:2,
                 root2:12:2);
260           END (* real roots *)
270         ELSE WRITELN('Roots are
             imaginary');
280       WRITE('next: ') {$w-};
290       READLN(a,b,c) {$w+};
300     END;(* a<>0.0 *)
310 END. (* exsqrt *)
```

Input:

```
3 19 20
```

Output:

```
Roots:        -1.33        -5.00
```

Syntax

[*unsigned-integer* ":"] *unlabeled-statement*

where *unlabeled-statement* is

"BEGIN" *statement* { ";" *statement* } "END"
|
assignment statement
|
IF statement
|
CASE statement
|
WHILE statement
|
REPEAT statement
|
FOR statement
|
procedure call
|
GOTO statement

Description

Statement	Use
Assignment	To give a value to a variable or a function
IF	To select one of two options
CASE	To select one of several options
WHILE, REPEAT, FOR	To execute statements repeatedly
Procedure	To begin the execution of a procedure
GOTO	To alter the flow of the program

STR

Syntax “STR” “(” *integer-expression* “,” *string-variable* “)”

Description The STR procedure converts an integer into its string representation. The value of *integer-expression* is represented as a string and stored in *string-variable*.

Example The following program reads an integer device number of a printer. The integer is converted into its string representation, used to open the device. A specified number of strings are entered from the keyboard and then printed.

```
100 PROGRAM exstr;
110 VAR filepr:TEXT;
120     strdev,strdata:STRING;
130     device,index,count:INTEGER;
140 BEGIN
150   WRITE('Enter device # of printer: ')
       {$w-};
160   READLN(device);
170   STR(device,strdev);
180   REWRITE(filepr,strdev);
190   WRITE('Enter number of lines: ');
200   READLN(count);
210   FOR index:=1 TO count DO
220     BEGIN
230       WRITE('? ');
240       READLN(strdata);
250       PAGE(filepr);
260       WRITELN(filepr,'Page ',index);
270       WRITELN(filepr,strdata);
280     END;(* for index *)
290 END. (* exstr *)
```

STRING

Syntax “STRING” { “[“ *integer-constant* “]” }

Description The predefined type STRING represents a series of characters enclosed in apostrophes. An apostrophe within a string is denoted by two apostrophes. A string's dynamic length is number of characters that were last assigned to it. The maximum length of a STRING item can be specified in a VAR declaration by including *integer-constant* in brackets after the word STRING. The maximum length allowed for a STRING type is 255 characters. When *integer-constant* is not specified, a default length of 80 characters is assumed for string.

A string can be assigned a value in an assignment statement or in an input statement.

The relational operator scan be used to compare strings. If characters in two strings are the same, the strings are equal. If the two strings have an unequal length, the comparison is performed as if the shorter string contained enough null characters (refer to appendix G) to make its length equal to the other string. Two strings are compared character by character according to each character's ASCII code. When the characters are different, the strings are ordered according to the ordering of the characters' ASCII codes.

The following functions and procedures can be used with strings.

CONCAT
COPY
DELETE
INSERT
LENGTH
POS
STR

subrange

Syntax *constant1* “..” *constant2*

Description A subrange is a subsequence of a series of ordinal values, where *constant1* specifies the first value and *constant2* specifies the last value in the series. *Constant1* must be less than or equal to *constant2*. The symbol .. denotes all the values between *constan1l* and *constant2*.

Subranges can be used in array declarations and in definitions of sets for use with the IN operator.

SUCC

Syntax "SUCC" "(" *ordinal-expression* ")"

Description The SUCC function returns the value that follows ordinal-expression in the set of values to which ordinal-expression belongs. lf ordinal-expression is the last value in its set, an error occurs when SUCC is executed.

Example The following program reads a string from the keyboard. Each character in the string is then changed to its successor and the string is displayed.

```
100 PROGRAM exsucc;
110 VAR count:INTEGER;
120     strval:STRING;
130 BEGIN
140   WRITELN('Have someone decode a
       message');
150   WRITE('Enter line of text: ') {$w-};
160   READLN(strval) {$w+};
170   FOR count:=l TO LENGTH(strval) DO
180     strval[count]:=SUCC(strval[count]);
190   WRITELN(strval);
200 END. (* exsucc *)
```

Input:

```
Bookkeeping is the exception
```

Output:

```
Cppllffqjoh!jt!uif!fydfqujpo
```

TEXT

Syntax “TEXT”

Description The predefined type TEXT is a file type that represents a file of CHAR. A TEXT file is divided into records, each of which ends with an end-of-line ENTER marker.

On a TEXT file, the EOLN function is TRUE when the next character to be read is the end-of-line marker and the EOF function is TRUE when the next character to be read is the end-of-file marker.

THEN

Syntax Refer to IF

Description The reserved word THEN is part of the IF statement. The statement following the word THEN is executed when the Boolean expression being tested in the IF statement is TRUE

If more than one statement is to be executed, the statement must be enclosed in the words BEGIN and END.

TO

Syntax Refer to FOR

Description The reserved word TO is used in the FOR statement to cause the control variable to take consecutive succeeding values in its defined set of values. For INTEGER types, the control variable is increased by 1 each time through the FOR loop. For a CHAR type, the control variable becomes the succeeding character each time through the loop.

TRUE

Syntax “TRUE”

Description The predefined BOOLEAN constant TRUE is equal to the BOOLEAN value TRUE. TRUE is defined to be greater than FALSE. The ordinal value of TRUE is 1.

TRUNC

Syntax “TRUNC” “(” *real-expression* “)”

Description The TRUNC function returns the integer portion of real-expression. The value returned is an integer;the fractional part of real-expression is discarded.

Example The following program reads a specified number of real values and computes their sum. The sum is then truncated and displayed.

```
100 PROGRAM extrunc;
110 VAR index,count:INTEGER;
120     money,donation:REAL;
130 BEGIN
140   money:=0.0;
150   WRITE('Enter # of donations: ')
       {$w-};
160   READLN(count);
170   FOR index:=1 TO count DO
180     BEGIN
190       WRITE(': ');
200       READLN(donation);
210       money:=money+donation;
220     END;(* donations *) {$w+}
230   WRITELN('Donations are around: $',
       TRUNC(money);
240 END. (* extrunc *)
```

TYPE

Syntax

"TYPE" *identifier* ":" *type* ";" { *identifier* ":" *type* ";" }

where *type* can be

"INTEGER"
|
"BOOLEAN"
|
"CHAR"
|
"STRING"
|
"REAL"
|
array-type
|
file-type
|
"TEXT"

Description

TYPE defines the set of values that an identifier or function can be assigned. The predefined types are INTEGER, REAL, STRING, CHAR, BOOLEAN, ARRAY OF, PACKED ARRAY OF, TEXT, and INTERACTIVE. New types can be defined with the TYPE declaration. The identifier being defined as a new type must be one of the predefined types.

The type of a constant is declared implicitly when it is defined in a CONST declaration. The type of an identifier must be declared explicitly in either a TYPE declaration or a VAR declaration. The TYPE declaration defines the type of data that the identifier can take.

UNBREAK

Syntax "UNBREAK" [*line-number* { "," *line-number*}]

Description The UNBREAK command removes all the breakpoints in a program that have been set by the BREAK command. Optionally, only the breakpoints for those lines specified are removed.

Example

```
UNBREAK
```

removes all breakpoints.

```
UNBREAK 250,375
```

removes the breakpoints set at lines 250 and 375

VAR

Syntax “VAR” *identifier* { “,” *idcntifier* } “:” *type* “;” { *identifier* { “.” *Identifier* } “:” *type* “;” }

Description The VAR declaration isused to name all the variables that used in a program. The interpreter reserves the memory required for each variable specified in a VAR declaration

Each CHAR and BOOLEAN type is allocated one byte. A STRING is allocated its declared length plus one for its byte. An INTEGER is allocated two bytes and a REAL is allocated eight bytes.

VERIFY

Syntax “VERIFY” “‘” *device* “.” *filename* “‘”

Description The VERIFY command determines whether data was saved to an external storage device or was loaded into memory correctly. VERIFY is used after a SAVE or OLD command to compare the program in memory to the program on the external storage device. If a difference is found, an error message is displayed. If no difference is found, the flashing cursor appears in column 1.

Device.filename identifies the device and the file in which the program is stored. *Device* is the number associated with the peripheral device and can be from 1 through 255. *Filename* identifies the file.

Example

```
SAVE '1.accounts'
```

saves the program in memory to device 1 under the filename `accounts`.

```
VERIFY '1.accounts'
```

verifies whether the program was stored correctly.

```
OLD '1.formula'
```

loads into memory the file named `formula` located on device 1.

```
VERIFY '1.formula'
```

verifies whether the file was loaded correctly.

WHILE

Syntax “WHILE” *Boolean-expression* “DO” *statement*

Description The WHILE statement executes *statement* repeatedly as long as *Boolean-expression* is true. *Statement* can be a statement or several statements. If several statement are to be executed in the WHILE loop, the reserved words BEGIN and END must be used to enclose them in a single compound statement.

Before statement is executed, *Boolean-expression* is evaluated to determine if it is true. Statement is executed repeatedly as long as *Boolean-expression* is true. When *Boolean-expression* becomes false, execution continues with the next statement. If Boolean-expression is false when WHILE is first executed, *statement* is not executed.

Example The following program reads real values and computes and displays their square roots until a non-positive number or zero is entered.

```
100 PROGRAM exwhile;
110 VAR realarg:REAL;
120 BEGIN
130   WRITE('Enter value: ) {$w-};
140   READLN(realarg) {$w+};
150   WHILE realarg>0.0 DO
160     BEGIN
170       WRITELN('Sq. root of ',realarg,
           ' is ',SQRT(realarg));
180       WRITE('next: ') {$w-};
190       READLN(realarg){$w+);
200     END;(* while argument valid *)
210 END. (* exwhile *)
```

WRITE

Syntax

“WRITE” “(” [*file-identifier* “,”] *output-item* { “,” *output-item* } “)”

where *output-item* is

string-expression [“:” *integer-expression*]
|
character-expression [“:” *integer-expression*]
|
integer-expression [“:” *integer-expression*]
|
real-expression [“:” *integer-expression* “:” *integer-expression*]

Description

The WRITE statement formats and writes data to the file specified by *file-identifier*. *File-identifier* must be the identifier of a file opened by REWRITE. If *file-identifier* is omitted, the file OUTPUT is assumed.

Output-item is a list of expressions, separated by commas, whose values are to be written. The output list is interpreted in order from left to right. After the last item is written, the WRITE statement leaves the cursor positioned at the end of the last data item. The next input/output statement can read or write more data from that position.

If a WRITE to the display is followed by a READ or READLN from the display, the cursor must not be positioned in the last column. If a displayed item moves the cursor past the last column and a READ or READLN attempts to input data from the display, the displayed item is truncated so that the cursor appears in the last column, waiting to accept data. Note that when WRITE is used for input prompts, the interpreter option wait (w) should be turned off ({$w-}). Otherwise, the ENTER key must be pressed before data can be entered.

An item enclosed in apostrophes is written as it appears inside the apostrophes. An apostrophe within a quoted item must be entered as two apostrophes. Any item not enclosed in apostrophes has its value written.

Refer to WRITELN for a description of how to format an output statement and how items are written unformatted.

Example

The following program uses the WRITE statement to display data on the same line. When the WRITE statement is used as a prompt, the interpreter option wait (w) is turned off so that the ENTER key does not have to be pressed before input can be entered.

```
100 PROGRAM exwrite;
110 VAR ch:CHAR;
120 BEGIN
130   WRITE('WRITE');
140   WRITE('adds the next output') {$w-};
150   WRITELN {$w+};
160   WRITE('to its line.');
170   WRITE(' Q');
180   WRITE('E');
190   WRITELN('D');
200   WRITE('Use WRITE for input prompts: ')
       {$w-};
210   READ(ch) {$w+};
220   WRITE(ch);
230 END. (* exwrite *)
```

Input:

```
C
```

Output:

```
WRITE adds the next output
to its line. QED
Use WRITE for input prompts :
C
```

WRITELN

Syntax

"WRITELN" ["(" [*file-identifier* ","] *output-item* { "," *output-item* } ")"]

where *output-item* is

string-expression [":" *integer-expression*]
¦
character-expression [":" *integer-expression*]
¦
integer-expression [":" *integer-expression*]
¦
real-expression [":" *integer-expression* ":" *integer-expression*]

Description

The WRITELN statement formats and writes data to the file specified by *file-identifier*. *File-identifier* must be the identifier of an open file. If file-identifier is omitted, the file OUTPUT is assumed. When a WRITELN to the display is executed, the interpreter option wait (w) should usually be turned on. Otherwise, the characters may be displayed too quickly to be read.

Output-item is a list of expressions, separated by commas, whose values are to be written. The output list is interpreted in order from left to right. After the last item is written, the WRITELN statement writes an end-of-line marker and advances to the first position of the next line.

An item enclosed in apostrophes is written as it appears inside the apostrophes. An apostrophe within a quoted item must be entered as two apostrophes. Any item not enclosed in apostrophes has its value written.

Numeric Formats
Integers are written without a decimal point;the maximum integer allowed is 32767 and the minimum integer allowed is -32768.

Real values can have a magnitude as small as ± 1.0E-128 or as large as ± 9.9999999999999E+127. Real values are written with at least one digit to the left of the decimal point and at least one digit to the right of the decimal point.Trailing zeros in the fractional part are omitted.

Real values are written in either decimal or scientific notation. A number is written in scientific notation when magnitude is 9999999999.49995 or greater. A real value is written in decimal notation with up to 10 significant digits. If the number has more than ten significant digits, the value is rounded to ten digits.

A real value written in scientific notation has the following form.

±*mantissa*E ±*exponent*

The mantissa is written with seven or fewer digits, with least one digit to the left and at least one digit to the right the decimal point. Trailing zeros in the fractional part are omitted.

The exponent is two or three digits preceded by a plus (+) or minus (-) sign. When the exponent is two digits, the mantissa is limited to seven digits. When the exponent is three digits, the mantissa is limited to six digits. When necessary, the mantissa is rounded to the appropriate number of digits.

Character Formats
An item that is a CHAR data type is displayed in one column. A STRING item is displayed in the number of columns required for the current length of the string.

Formatting
An item is formatted when a colon is written immediately after it, followed by *integer-expression*. *Integer-expression* is the field-width specification or the number of columns used to write the item. If the number of columns required to write an item is less than the field-width specification, the item is written right-justified with leading blanks.

The maximum field-width specifications for TI-74 Pascal is shown in the table on the next page.

Data type	Maximum Field-Width Specification
INTEGER	80
REAL	14
CHAR	80
STRING	80

You can display a string with a length greater than 80 by using no format specification. The string is displayed 80 characters at a time. The ENTER key can be pressed to display the next 80 characters in the string until the end of the string is reached.

For integer values, the field-width specification must allow a column for a negative sign. For real values, the field-width specification must allow for the decimal point and a negative sign when there is one.

For real data, a colon followed by an integer expression can appear after the field-width specification. This second integer expression indicates the number of digits to be written after the decimal point. If the second specification is omitted, the real value is written right-justified in the number of columns specified by the field-width specification.

A real value that is formatted is written according to the following conventions:

Digits to the left of the decimal point	Result
Less than 10 digits	Written with all significant digits according to the format specifications.
From 10 through 14 digits	Written in decimal notation with all significant digits. If the field-width specification is larger than the number of digits, the value is right-justified.
More than 14 digits	Written in scientific notation with 13 or 14 digits.

If the field-width specification is less than the number of columns required, an item is written in the number of columns required to write its values.

Example

The following program uses the WRITELN statement to display messages on successive lines. The values of intege reals (both decimal form and scientific notation), and characters are then displayed.

```
PROGRAM exwritln;
BEGIN
  WRITELN('WRITELN writes its data');
  WRITELN(' and moves to the first
   column');
  WRITELN(' of the next line');
  WRITELN('C');
  WRITELN;
  WRITELN(545:15);
  WRITELN(867.5309:12);
  WRITELN(867.5309:10:3);
  WRITELN(86753098675310.0:14);
  WRITELN(86753098675310.0:14:2);
  WRITELN(8.675309867531E+14);
  WRITELN(8.675309867531E+14:14:2);
END. (* exwritln *)
```

Output:

```
WRITELN writes its data
 and moves to the first column
 of the next line
C
            545
    867.5309
   867.531
86753098675310.0
86753098675310.0
8.67531E+14
8.675309867531E+14
```

Appendix A—Commands

This appendix lists all the commands available in TI-74 Pascal. Commands may not appear in a Pascal program. The acceptable abbreviation for a command appears in parentheses.

BREAK
BYE
CONTINUE (CON)
DEL
FORMAT
LIST
NEW
NUMBER (NUM)
OLD
RENUMBER (REN)
RUN
SAVE
UNBREAK
VERIFY

Appendix B—Functions

This appendix lists the functions that are available in TI-74 Pascal. Note that a function cannot appear by itself as an imperative: it can, however, appear in a statement that is used as an imperative.

ABS
ATAN
CHR
CONCAT
COPY
COS
EOF
EOLN
EXP
IORESULT
LENGTH
LN
LOG
MEMAVAIL
ODD
ORD
POS
PRED
PWROFTEN
ROUND
SCAN
SIN
SIZEOF
SQR
SQRT
SUCC
TRUNC

Appendix C—Statements

This appendix lists the predefined statements in TI-74 Pascal. The statements are grouped according to use. The statements that cannot be used as imperatives (executed immediately rather than in a program) are printed with an asterisk (*).

Loop Statements	FOR REPEAT WHILE
Input/Output Statements	GOTOXY READ READLN WRITE WRITELN
Branching Statements	CASE GOTO* IF
File Procedures	CLOSE PAGE RESET REWRITE
Array Procedures	FILLCHAR MOVELEIT MOVERIGHT
String Procedures	DELETE INSERT STR
Termination Procedures	EXIT* HALT*

Appendix D—Procedures/Functions

This appendix lists the predefined procedures and functions available in Pascal. The routines are grouped according to use.

Numerical Functions	ABS ATAN COS EXP LN LOG ODD	PWROFTEN ROUND SIN SQR SQRT TRUNC
String Procedures/ Functions	CONCAT COPY DELETE	LINSERT LENGTH POS
Memory Functions	MEMAVAIL SIZEOF	
Array Procedures/ Functions	FILLCHAR MOVELEFT	MOVERIGHT SCAN
Ranking Functions	ORD PRED	SUCC
Input/Output Procedures/ Functions	CLOSE EOF EOLN GOTOXY IORESULT PAGE	READ READLN RESET REWRITE WRITE WRITELN
Program Terminating Procedures	EXIT HALT	
Conversion Procedures/ Functions	CHR STR	

Appendix E—Declarations and Data Types

The declarations available in TI-74 Pascal are listed below in the order in which they must appear in a program.

LABEL declaration
CONSTant declaration
TYPE declaration
VARiable declaration
FUNCTION/PROCEDURE declaration including FORWARD declaration

The data types available in TI-74 Pascal are the following.

ARRAY OF type
BOOLEAN
CHAR
INTEGER
PACKED ARRAY OF type
REAL
STRING
TEXT

Appendix F—Reserved Words

This appendix contains an alphabetical list of the reserved words in TI-74 Pascal. These words have pre-defined meanings and cannot be used as a user-defined identifier. Reserved words are printed or displayed in uppercase characters.

ABS
ALL
AND
ARRAY
ATAN
BEGIN
BOOLEAN
BREAK
BYE
CASE
CHAR
CHR
CLOSE
CON
CONCAT
CONST
CONTINUE
COPY
COS
DEL
DELETE
DIV
DO
DOWNTO
ELSE
END
EOF
EOLN
EXIT
EXP
FALSE
FILE
FILLCHAR
FOR
FORMAT
FORWARD
FUNCTION
GOTO
GOTOXY
HALT
IF
IN
INPUT
INSERT
INTEGER
INTERACTIVE
IORESULT
KEYBOARD
LABEL
LENGTH
LIST
LN
LOCK
LOG
MAXINT
MEMAVAIL
MOD
MOVELEFT
MOVERIGHT
NEW
NOT
NUM
NUMBER
ODD
OF
OLD
OR
ORD
OUTPUT
PACKED
PAGE
POS
PRED
PROCEDURE
PROGRAM
PROTECTED
PURGE
PWROFTEN
READ
READLN
REAL
REN
RENUMBER
REPEAT
RESET
REWRITE
ROUND
RUN
SAVE
SCAN
SIN
SIZEOF
SQR
SQRT
STR
STRING
SUCC
TEXT
THEN
TO
TRUE
TRUNC
TYPE
UNBREAK
UNTIL
VAR
VERIFY
WHILE
WRITE
WRITELN

Appendix G—ASCII Codes and Keycodes List

The following table lists the ASCII character codes in decimal and hexadecimal notation. The ASCII codes produced and/or character{s) displayed when the key or key sequence is pressed are shown in the column titled Character. The characters that can be displayed using the CHR function are shown in the column titled Displayed Using CHR. The keys pressed to generate the ASCII code are shown in the column titled Key Sequence.

System-reserved character codes (0-15) and the user-assigned keys (codes 128-137) are shown as two asterisks (**).

ASCII Code Dec	Hex	Character	Displayed Using CHR	Key Sequence
00	00	NULL	**	CTL 0
01	01	SOH	**	CTL A
02	02	STX	**	CTL B
03	03	ETX	**	CTL C
04	04	EOT	**	CTL D
05	05	ENQ	**	CTL E
06	06	ACK	**	CTL F
07	07	BEL	**	CTL G
08	08	BS	**	CTL H
09	09	HT	**	CTL I
10	0A	LF	**	CTL J
11	0B	VT	**	CTL K
12	0C	FF	**	CTL L
13	0D	CR	**	CTL M or ENTER
14	0E	SO	**	CTL N
15	0F	SI	**	CTL O
16	10	DLE		CTL P
17	11	DCl		CTL Q
18	12	DC2		CTL R
19	13	DC3		CTL S
20	14	DC4		CTL T
21	15	NAK		CTL U
22	16	SYN		CTL V
23	17	ETB		CTL W

ASCII Code Dec	Hex	Character	Displayed Using CHR	Key Sequence
24	18	CAN		CTL X
25	19	EM		CTL Y
26	1A	SUB		CTL Z
27	1B	ESC		CTL CLR
28	1C	FS		CTL +/-
29	1D	GS		CTL ;
30	1E	RS		CTL .
31	1F	US		CTL ,
32	20	SPACE	SPACE	SPACE
33	21	!	!	SHIFT 1
34	22	"	"	SHIFT 2
35	23	#	#	SHIFT 3
36	24	$	$	SHIFT 4
37	25	%	%	SHIFT ,
38	26	&	&	SHIFT 5
39	27	'	'	SHIFT SPACE
40	28	(	(	SHIFT ↑
41	29	)	)	SHIFT ↓
42	2A	*	*	*
43	2B	+	+	+
44	2C	,	,	,
45	2D	-	-	-
46	2E	.	.	.

ASCII Code			Displayed	Key
Dec	Hex	Character	Using CHR	Sequence
47	2F	/	/	/
48	30	0	0	0
49	31	1	1	1
50	32	2	2	2
51	33	3	3	3
52	34	4	4	4
53	35	5	5	5
54	36	6	6	6
55	37	7	7	7
56	38	8	8	8
57	39	9	9	9
58	3A	:	:	SHIFT ;
59	3B	;	;	;
60	3C	<	<	SHIFT 0
61	3D	=	=	SHIFT ENTER
62	3E	>	>	SHIFT .
63	3F	?	?	SHIFT +/-
64	40	@	@	CTL 2
65	41	A	A	SHIFT A
66	42	B	B	SHIFT B
67	43	C	C	SHIFT C
68	44	D	D	SHIFT D
69	45	E	E	SHIFT E
70	46	F	F	SHIFT F

ASCII Code Dec	Hex	Character	Displayed Using CHR	Key Sequence
71	47	G	G	SHIFT G
72	48	H	H	SHIFT H
73	49	I	I	SHIFT I
74	4A	J	J	SHIFT J
75	4B	K	K	SHIFT K
76	4C	L	L	SHIFT L
77	4D	M	M	SHIFT M
78	4E	N	N	SHIFT N
79	4F	O	O	SHIFT O
80	50	P	P	SHIFT P
81	51	Q	Q	SHIFT Q
82	52	R	R	SHIFT R
83	53	S	S	SHIFT S
84	54	T	T	SHIFT T
85	55	U	U	SHIFT U
86	56	V	V	SHIFT V
87	57	W	W	SHIFT W
88	58	X	X	SHIFT X
89	59	Y	Y	SHIFT Y
90	5A	Z	Z	SHIFT Z
91	5B	[	[	CTL 8
92	5C	¥	¥	CTL /
93	5D	]	]	CTL 9
94	5E	^	^	SHIFT 6

ASCII Code Dec	Hex	Character	Displayed Using CHR	Key Sequence
95	5F	_	_	CTL 5
96	60	`	`	CTL 3
97	61	a	a	A
98	62	b	b	B
99	63	c	c	C
100	64	d	d	D
101	65	e	e	E
102	66	f	f	F
103	67	g	g	G
104	68	h	h	H
105	69	i	i	I
106	6A	j	j	J
107	6B	k	k	K
108	6C	l	l	L
109	6D	m	m	M
110	6E	n	n	N
111	6F	o	o	O
112	70	p	p	P
113	71	q	q	Q
114	72	r	r	R
115	73	s	s	S
116	74	t	t	T
117	75	u	u	U
118	76	v	v	V

ASCII Code Dec	Hex	Character	Displayed Using CHR	Key Sequence
119	77	w	w	W
120	78	x	x	X
121	79	y	y	Y
122	7A	z	z	Z
123	7B	{	{	CTL 6
124	7C	¦	d	CTL 1
125	7D	}	e	CTL 7
126	7E	→	→	CTL 4
127	7F	←	←	SHIFT →
128	80	**		FN 0
129	81	**		FN 1
130	82	**		FN 2
131	83	**		FN 3
132	84	**		FN 4
133	85	**		FN 5
134	86	**		FN 6
135	87	**		FN 7
136	88	**		FN 8
137	89	**		FN 9
138	8A			
139	8B			
140	8C			
141	8D			SHIFT /
142	8E			SHIFT '

ASCII Code Dec	Hex	Character	Displayed Using CHR	Key Sequence
143	8F			SHIFT -
144	90			SHIFT +
145	91			CTL *
146	92			CTL -
147	93			CTL +
148	94	NEW		FN ←
149	95	NUMBER		FN …
150	96	RENUMBER		FN ↑
151	97	MEMAVAIL		FN ↓
152	98	VERIFY		FN /
153	99	SAVE		FN *
154	9A	OLD		FN -
155	9B	LIST		FN +
156	9C	BOOLEAN		FN .
157	9D	CHAR		FN ,
158	9E	LABEL		FN ;
159	9F	WHILE		FN +/-
160	A0	DELETE		FN CLR
161	A1	FOR	｡	FN A
162	A2	READ(	｢	FN B
163	A3	REWRITE(	｣	FN C
164	A4	DOWNTO	､	FN D
165	A5	ATAN(	･	FN E
166	A6	IF	ｦ	FN F

ASCII Code Dec	Hex	Character	Displayed Using CHR	Key Sequence
167	A7	THEN	ｧ	FN G
168	A8	ELSE	ｨ	FN H
169	A9	SQRT	ｩ	FN I
170	AA	BEGIN	ｪ	FN J
171	AB	PROCEDURE	ｫ	FN K
172	AC	FUNCTION	ｬ	FN L
173	AD	REAL	ｭ	FN M
174	AE	INTEGER	ｮ	FN N
175	AF	REPEAT	ｯ	FN O
176	B0	UNTIL	ｰ	FN P
177	B1	SIN(	ｱ	FN Q
178	B2	ROUND(	ｲ	FN R
179	B3	TO	ｳ	FN S
180	B4	LN(	ｴ	FN T
181	B5	EXP(	ｵ	FN U
182	B6	READLN	ｶ	FN V
183	B7	COS(	ｷ	FN W
184	B8	WRITE(	ｸ	FN X
185	B9	LOG(	ｹ	FN Y
186	BA	WRITELN(	ｺ	FN Z
187	BB	BREAK	ｻ	FN BREAK
188	BC		ｼ	SHIFT RUN
189	BD		ｽ	
190	BE	CONTINUE	ｾ	FN RUN

ASCII Code Dec	 Hex	 Character	Displayed Using CHR	Key Sequence
191	BF	RUN	ｿ	RUN
192	C0		ﾀ	
193	C1		ﾁ	
194	C2		ﾂ	
195	C3		ﾃ	
196	C4		ﾄ	
197	C5		ﾅ	
198	C6		ﾆ	+/-
199	C7		ﾇ	
200	C8		ﾈ	
201	C9		ﾉ	
202	CA		ﾊ	
203	CB		ﾋ	
204	CC		ﾌ	
205	CD		ﾍ	
206	CE		ﾎ	
207	CF		ﾏ	
208	D0		ﾐ	SHIFT FN 0
209	D1		ﾑ	SHIFT FN 1
201	D2		ﾒ	SHIFT FN 2
211	D3		ﾓ	SHIFT FN 3
212	D4		ﾔ	SHIFT FN 4
213	D5		ﾕ	SHIFT FN 5
214	D6		ﾖ	SHIFT FN 6

ASCII Code Displayed Key

Dec	Hex	Character	Using CHR	Sequence
215	D7		ﾗ	SHIFT FN 7
216	D8		ﾘ	SHIFT FN 8
217	D9		ﾙ	SHIFT FN 9
218	DA		ﾚ	
219	DB		ﾛ	
220	DC		ﾜ	
221	DD		ﾝ	
222	DE		ﾞ	
223	DF		ﾟ	
224	E0		α	
225	E1		ä	SHIFT -
226	E2		β	
227	E3		ε	
228	E4		μ	
229	E5	PB	σ	SHIFT 9
230	E6	OFF	ρ	OFF
231	E7	BREAK	ç	BREAK
232	E8	UP	√	↑
233	E9	DOWN	¬	↓
234	EA		¡	
235	EB		×	
236	EC		¢	
237	ED		£	

ASCII Code Dec	Hex	Character	Displayed Using CHR	Key Sequence
238	EE		n̄	
239	EF		ö	
240	F0		p	MODE
241	F1		q	
242	F2		θ	
243	F3		∞	
244	F4		Ω	
245	F5		ü	CTL RUN
246	F6	DEL	Ω	SHIFT 7
247	F7	INS	π	SHIFT 8
248	F8	HOME	x̄	CTL ↑
249	F9	DELREST	¿	CTL ↓
250	FA	CLR	千	CLR
251	FB	BTAB	万	CTL ←
252	FC	LEFT	円	←
253	FD	FTAB	÷	CTL →
254	FE	RIGHT		→
255	FF		█	CTL SPACE

Appendix H—Accuracy Information

Calculation Accuracy

The TI-74, like all computers, operates within preset limits with a fixed set of rules. The mathematical tolerance of the computer is controlled by the number of digits it uses for calculations.

The TI-74 uses a minimum of 13 digits to perform calculations. The results are rounded to 10 digits when displayed in the default display format. The computer's 5/4 rounding technique adds 1 to the least significant digit of the display if the next nondisplayed digit is five or more. If this digit is less than five, no rounding occurs. Without these extra digits, inaccurate results such as the following would frequently be displayed.

```
1/3 x 3 = 0.9999999999
```

This result occurs because 1/3 is maintained as 0.3333333333 in the finite internal representation of a number. However, when 1/3x3 is rounded to 10 digits, the answer 1.0 is displayed.

The more complex mathematical functions are calculated using iterative and polynomial methods. The cumulative rounding error is usually kept beyond the tenth digit so that displayed values are accurate. Normally there is no need to consider the undisplayed digits. However, certain calculations may cause the unexpected appearance of these extra digits as shown below.

2/3 ~ 0.66666666666667 and 1/3 = 0.33333333333333

2/3 -1/3 -1/3 = 0.00000000000001 (displayed 1.E-14)

Such possible discrepancies in the least significant digits of a calculated result are important when testing if a calculated result is equal to another value. In testing for equality, precautions should be taken to prevent improper evaluation.

A useful technique is to test whether two values are sufficiently close together (rather than absolutely equal) as shown below.

Instead of `IF X=Y THEN`

use `IF ABS(X-Y)<1.0E-11 THEN`

Internal Numeric Representation

The TI-74 uses radix-100 format for internal calculation. A single radix-100 digit ranges in value from 0 to 99 in base 10. The computer uses a 7-digit mantissa which results in 13 to digits of decimal precision. A radix-100 exponent ranges in value from -64 to +63, which yields decimal exponents from 10-128 to 10+127. The exponent and the 7-digit mantissa combine to provide a decimal range from -9.9999999999999E+127 through -1.0E-128;zero;and then +1.0E-128 through +9.9999999999999E+127.

The internal representation of the radix-l00 format requires eight bytes. The first byte contains the exponent and the algebraic sign of the entire floating point number. The exponent is a 7-bit hexadecimal (base 16) value offset or biased by 40_{16} (the subscript 16 indicates hexadecimal values in this appendix). The correspondence between exponent values is shown below.

Biased hexadecimal value	00_{16}	to	40_{16}	to	$7F_{16}$
Radix-l00 value	-64	to	0	to	+63
Decimal value	-128	to	0	to	+126

If the floating point number is negative, the first byte (the exponent value) is inverted (1's complement). Each byte of the mantissa contains a radix-100 digit from 0 to 99 represented in binary coded decimal (BCD) form. In other words, the most significant four bits of each byte represents a decimal digit from 0 to 9 and the least significant four bits represents a decimal digit from 0 to 9. The first byte of the mantissa contains the most significant digit of the radix-100 number. The number is normalized so that the decimal point immediately follows the most significant radix-100 digit. If the first byte of the mantissa contains 0, the floating point number is zero, regardless of the contents of the remaining bytes.

The examples on the next page show some decimal values and their internal representation.

Decimal	Internal							
1.0	4°$_{16}$	01$_{16}$	°°$_{16}$	°°$_{16}$	°°$_{16}$	°°$_{16}$	°°$_{16}$	°°$_{16}$
10.0	4°$_{16}$	10$_{16}$	°°$_{16}$	°°$_{16}$	°°$_{16}$	°°$_{16}$	°°$_{16}$	°°$_{16}$
100.0	41$_{16}$	01$_{16}$	°°$_{16}$	°°$_{16}$	°°$_{16}$	°°$_{16}$	°°$_{16}$	°°$_{16}$
1234.0	41$_{16}$	12$_{16}$	34$_{16}$	°°$_{16}$	°°$_{16}$	°°$_{16}$	°°$_{16}$	°°$_{16}$
pi	4°$_{16}$	03$_{16}$	14$_{16}$	15$_{16}$	92$_{16}$	65$_{16}$	35$_{16}$	9°$_{16}$
-pi	BF$_{16}$	03$_{16}$	14$_{16}$	15$_{16}$	92$_{16}$	65$_{16}$	35$_{16}$	9°$_{16}$
e	4°$_{16}$	20$_{16}$	71$_{16}$	82$_{16}$	81$_{16}$	82$_{16}$	84$_{16}$	6°$_{16}$

where pi has a value of 3.14159265359 and *e* has a value of 2.71828182846.

Appendix I—Error Messages

The following is a list of the error messages generated by the TI-74 Pascal interpreter. The first list, arranged Alphabetically by message, provides information about the probable cause of the error. The second list, arranged in ascending order by error code, serves as a cross reference to ocate the message associated with a particular error code. The input/output (I/O) codes are listed last.

When an error message is displayed, the ←, →, ↑, ↓ and **SHIFT PB** keys can be used to display additional system error information and to edit an erroneous line.

SHIFT PB	is used when an error occurs after a line is entered. **SHIFT PB** displays the erroneous entry, which can then be edited and entered again.
→	is used when an error occurs during program execution. → displays the error code and the line number of the line being executed (where the error occurred) in an `E`*nnnn* `L`*mmmmm* format where *nnnn* is the error code and *mmmmm* is the line number. (This line is not necessarily the one that is the source of the problem because an error may occur because values generated or actions taken elsewhere the program.) When an I/O error occurs during program execution, → displays the error code, the filename, and line number in the format `E0,`*zzz* *xxxxxxxx* `L`*yyyyy*, where *zzz* is the I/O error code, *xxxxxxxx* is the filename, and *yyyyy* is the line number of the line that was executing when the error occurred. When an I/O error occurs during execution of a command, the → key displays the error code, the I/O error code, and the device number in the format `E0`, *zzz* '*yyy*' format, where *zzz* is the I/O error code and *yyy* is the device number.
←	can be used to redisplay the error message immediately after the → key has been pressed-
↑ or ↓	is used when an error occurs during program execution to display the program line that was executing when the error occurred.

Messages Listed Alphabetically

Note: Codes preceded by an asterisk (*) are warnings and are recoverable; press the ENTER key to resume program execution. After a breakpoint, however, the CON command must be entered to resume program execution.

Code	Message/Cause
602	Bad program type
17	'BEGIN' expected
*600	Break
203	Constant exceeds range
155	Control variable must be local
300	Division by zero
54	'DO' expected
13	'END' expected
306	End of comment missing
18	Error in declaration part
125	Error in parameter list
10	Error in type
259	Expression too complicated
133	File comparison not allowed
100	File error
307	Format too big
101	Identifier declared twice
2	Identifire expected
56	'IF' expected
28	Illegal as imperative

400	Illegal character in text
143	Illegal control variable type
99	Illegal FORWARD declearation
412	Illegal line number
27	Illegal nesting
126	Illegal number of parameters
30	Illegal number of subscripts
29	Illegal operator
142	Illegal parameter solution
144	Illegal type of expression
134	Illegal type of operand
398	Implementation restriction
139	Incompatible index type
15	INTEGER expected
9998	Internal system error
0	I/O error
411	Line too long
413	Line not found
102	Low bound exceeds high bound
166	Multideclared label
165	Multidefined label
601	No breakpoint active
8	'OF' expected
135	Operand type must be BOOLEAN

9999	Out of memory
*604	Pascal System Initialized
603	Programmed halt
415	Protection violiation
201	REAL number error
25	Statement expected
33	Syntax error
52	'THEN' expected
55	'TO' or 'DOWNTO' expected
*414	Truncation warning
145	Type Conflict
104	Undeclared identifier
167	Undeclared label
168	Undefined label
401	Unexpected end of input
117	Unsatisfied FORWARD reference
53	'UNTIL' expected
303	Value is out of bounds
416	Value stack underflow
4	')' expected
5	':' expected
9	'(' expected
14	';' expected
16	'=' expected

51	':=' expected
26	',' expected
11	'[' expected
12	']' expected

Messages Listed in Numerical Order

Code	Message/Cause
0	I/O error
2	Identifier expected
4	')' expected
5	':' expected
8	'OF' expected
9	'(' expected
10	Error in type
11	'[' expected
12	']' expected
13	'END' expected
14	';' expected
15	INTEGER expected
16	'=' expected
17	'BEGIN' expected
18	Error in declaration part
25	Statement expected
26	',' expected
27	Illegal nesting

28	Illegal as imperative
29	Illegal operator
30	Illegal number of subscripts
33	Syntax error
51	':=' expected
52	'THEN' expected
53	'UNTIL' expected
54	'DO' expected
55	'TO' or 'DOWNTO' expected
56	'IF' expected
99	Illegal FORWARD declaration
100	File error
101	Identifier declared twice
102	Low bound exceeds high bound
104	Undeclared identifier
117	Unsatisfied FORWARD reference
125	Error in parameter list
126	Illegal number of parameters
133	File comparison not allowed
134	Illegal type of operand
135	Operand type must be BOOLEAN
139	Incompatible index type
142	Illegal parameter solution
143	Illegal control variable type

144	Illegal type of expression
145	Type Conflict
155	Control variable must be local
165	Multidefined label
166	Multideclared label
167	Undeclared label
168	Undefined label
201	REAL number error
203	Constant exceeds range
259	Expression too complicated
300	Division by zero
303	Value is out of bounds
306	End of comment missing
307	Format too big
398	Implementation restriction
400	Illegal character in text
401	Unexpected end of input
411	Line too long
412	Illegal line number
413	Line not found
*414	Truncation warning
415	Protection violiation
416	Value stack underflow
*600	Break

601	No breakpoint active
602	Bad program type
603	Programmed halt
*604	Pascal System Initialized
9998	Internal system error
9999	Out of memory

I/O Error Codes

The following list details the standard input/output (I/O) error codes. I/O error codes are displayed in one of the following forms:

- `I/O error` *ccc file-identifier*
- `I/O error` *ccc 'ddd'*

Where *ccc* is the I/O error code listed below, *file-identifier* is the file identifier used to RESET or REWRITE the file, and *ddd* is the device code associated with the peripheral device.

Code	Definition
1	**`DEVICE/FILE OPTIONS ERROR`** • Incorrect or invalid opton specified in '*device filename*'. • *Filename* too long or missing in '*device filename*'.
2	**`ERROR IN ATTRIBUTES`** • A RESET statement attempted to open a file that was not a text file.
3	**`FILE NOT FOUND`** • The file specified in one of the following operations does not exist. – RESET statement for a nonexistent file – OLD '*device filename*' – RUN '*device filename*' – DEL '*device filename*'

Code	Definition
4	**DEVICE,/FILE NOT OPEN** • Attempted to access a closed file with a READ, READLN,WRITE, WRITELN, or CLOSE statement. • File specified in EOF function is closed.
5	**DEVICE/FILE ALREADY OPEN** • Attempted to RESET, REWRITE, or DEL an open file.
6	**DEVICE ERROR** • A failure has occurred in the peripheral. Try the operation again.
7	**END OF FILE** • Attempted to read past the end of the file.
8	**DATA,/FI LE TOO LONG** • Attempted to output a record longer than the capacity of the device. • A file exceeded the maximum file length for a device.
9	**WRITE PROTECT ERROR** • Attempted to FORMAT a write-protected storage medium. • Attempted to REWRITE a write-protected file. • Attempted to DEL or CLOSE (and delete) a file from a write-protected medium.
10	**NOT REQUESTING SERVICE** • Response to a service request poll when the specified device did not request service. (This code is used in special applications and should not be encountered during normal execution of Pascal programs.)
11	**DIRECTORY FULL** • Attempted to REWRIIE or SAVE a new file on a device whose directory is full.
12	**BUFFER SIZE ERROR** • The VERIFY command found the program in memory was smaller than the program on the storage medium.

Code	Definition
13	**UNSUPPORTED COMMAND** • Attempted an operation not supported by the peripheral.
14	**DEVICE/FILE NOT OPENED FOR OUTPUT** • Attempted to write to a file or device opened for input.
15	**DEVICE/FILE NOT OPENED FOR INPUT** • Attempted to read from a file or device opened for output.
16	**CHECKSUM ERROR** • The checksum calculated on the input record was incorrect. Try the operation again.
20	**OUTPUT MODE NOT SUPPORTED** • Device specified in OPEN statement does not support output mode.
21	**INPUT MODE NOT SUPPORTED** • Device specified in RESET statement does not support input mode.
24	**VERIFY ERROR** • Program or data in memory does not match specified program or storage medium. Try the VERIFY command again. If the error is not cleared, repeat the entire sequence of storage or retrieval followed by verification.
25	**LOW BATTERIES IN PERIPHERAL** • Attempted an I/O operation iwht a device wihose batteries are low. Recharge the batteries in the printer/plotter.
26	**UNITIALIZED MEDIUM** • Attempted to open a file on an uninitialized storage medium. • Attempted to open a file on a storage medium which has been accidentally erased or destroyed.
32	**MEDIUM FULL** • No available space on storage medium

Code	Definition
255	**`TIME-OUT ERROR`** • Lost communication with the specified device. • Specified device is not connected to the I/O bus. **Note**: Check the cable connections and make sure that you are using the correct *device-number*.

Service Information

If you experience a problem with the cartridge, you can call or write Consumer Relations to discuss the problem.

For Service and General Information

If you have questions about service or the general use of your cartridge, please call Consumer Relations at:

1-806-747-1882

Please note that this is a toll number, and collect calls are not accepted.

You may also write to the following address:

Texas Instruments Incorporated
Consumer Relations
P.O. Box 53
Lubbock, Texas 79408

Please contact Consumer Relations:

⇒ Before returning the cartridge for service

⇒ For general information about using the cartridge

For Technical Information

If you have technical questions about the operation of the product or programming applications, write to Consumer Relations at the address given above, or call l-806-74l-2663. Please note that this is a toll number, and collect calls are not accepted.

Express Service

Texas Instruments offers an express service option for fast return delivery. Please call Consumer Relations for information.

Calculator Accessories

If you are unable to purchase calculator accessories (such as carrying cases or adapters) from your local dealer, you may order them from Texas Instruments. Please call Consumer Relations for information.

Returning Your Cartridge for Service

A defective cartridge will be either repaired or replaced with the same or comparable reconditioned model (at TI's option) when it is returned, postage prepaid, to at Texas Instruments Service Facility.

Texas Instruments cannot assume responsibility for loss or damage during incoming shipment. For your protection, carefully package the cartridge for shipment and insure it with the carrier. Be sure to enclose the following items with your cartridge:

- ⇒ Your full return address
- ⇒ Any accessories related ot the problem
- ⇒ A note describing the problem you experienced
- ⇒ A copy of your sales receipt or other proof of purchase to determine warranty status.

Please ship the cartridge postage prepaid;COD shipments cannot be accepted.

In-Warranty Service

For a cartridge covered under the warranty period, no charge is made for service.

Out-of-Warranty Service

A flat-rate charge by model is made for out-of-warranty service. To obtain the service charge for a particular model, call Consumer Relations before returing the product for service. (We cannot hold products in the Service Facility while providing charge information.)

Texas Instruments Service Facilities

U.S. Residents
(U.S. Postal Service)
Texas Instruments
P.O. Box 2500
Lubbock, Texas 79408

U.S. Residents
(other carriers)
Texas Instruments
2305 N. University
Lubbock, Texas 79415

Canadian Residents Only
Texas Instruments
41 Shelley Road
Richmond Hil, Ontario, L4C5G4

One-Year Limited Warranty

This Texas Instruments software cartridge warranty extends to the original consumer purchaser of the product.

Warranty Duration

This cartridge is warranted to the original consumer purchaser for a period of one (1) year from the original purchase date.

Warranty Coverage

This cartridge is warranted against defective materials or workmanship. This warranty covers the electronic and case components of the software cartridge. These components include all semiconductor chips and devices, plastics, boards, wiring, and all other hardware contained in this cartridge ("the Hardware"). This limited warranty does not extend to the programs contained in the cartridge and the accompanying book materials ("the Programs"). **The warranty is void if the cartridge has been damaged by accident, unreasonable use, neglect, improper service, or other causes not arising out of defects in material or workmanship.**

Warranty Disclaimers

Any implied warranties arising out of this sale, including but not limited to the implied warranties of merchantability and fitness for a particular purpose, are limited in duration to the above one-year period. Texas Instruments shall not be liable for loss ofuse ofthe cartridge or other incidental or consequential costs, expenses, or damages incurred by the consumer or any other user.

Some states do not allow the exclusion or limitations of implied warranties or consequential damages, so the above limitations or exclusions may not apply to you.

Legal Remedies

This warranty gives you specific legal rights, and you may also have other rights that vary from state to state.

Warranty Performance

During the above one-year warranty period, a defective TI cartridge will either be repaired or replaced with a reconditioned comparable model (at TI's option) when the cartridge is returned, postage prepaid, to a Texas Instruments Service Facility. The repaired or replacement cartridge will be in warranty for the remainder of the original warranty period or for six months, whichever is longer. Other than the postage requirement, no charge will be made for such repair or replacement. Texas Instruments strongly recommends that you insure the product for value prior to mailing.